UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

LARCH: UMA ALTERNATIVA PARA ESPECIFICAÇÃO FORMAL

Por

Ausberto S. Castro Vera

Dissertação submetida como requisito parcial para
a obtenção do grau de Mestre em
Ciência da Computação

Prof. Dr. Daltro J. Nunes
Orientador

Porto Alegre, 7 de maio de 1990

CATALOGAÇAO NA FONTE

Castro V., Ausberto S.
LARCH: Uma alternativa para especificação Formal. Porto Alegre, CPGCC- UFRGS, 1990.
iv.
Diss. (mestr. ci. comp) UFRGS-CPGCC, Porto Alegre, BR-RS, 1990.

Dissertação: Linguagens de Especificação Formal : LARCH : Especificação Formal : métodos e linguagens.

Dedico este trabalho
A Elba e Weyden Daniel

AGRADECIMENTOS

A Deus pelas bençãos recebidas.

A minha esposa Elba, pelo amor, incentivo, e compreensão recebidos sempre.

A meus pais e irmás: Sindulfo e Hilaria, Baltazar e Noema, Elizabeth e Delicia, pela ajuda e apoio recebidos no Perú.

Ao Professor Daltro J. Nunes, pela orientação deste trabalho e de outras disciplinas.

Ao corpo docente do Curso de Pós-Graduação em Ciência da Computação da UFRGS.

Aos colegas da turma 87, pelo constante apoio, as valiosas sugestões e críticas oportunas.

Ao CNPq e CAPES, pelo auxilio financeiro, recebido em forma de bolsa, durante o Curso de Pós-Graduação.

ESPECFICAÇAO

Faze uma arca de tábuas de cipreste; nela **farás** compartimentos, e a calafetarás com betume por dentro e por fora.

Dêste modo a farás: de trezentos côvados será o comprimento, de cinquenta a largura, e a altura de trinta.

Farás ao seu redor uma abertura de um côvado de alto; a porta da arca **colocarás** lateralmente; **farás** pavimentos na arca: um em baixo, um segundo e um terceiro.

Gênesis 6:14-16

Tudo tem o seu tempo determinado, e há tempo para todo propósito debaixo do firmamento:

Há tempo de apresentar, e tempo de concluir; tempo de **especificar**, e tempo para implementar o especificado, ...

Eclesiastes 3:1-8 (modificado)

SUMARIO

LISTA DE FIGURAS

RESUMO

Pesquisas recentes na área de especificação são enfáticas no uso prático de especificações formais no projeto de programas. Uma maneira de satisfazer isto, é providenciando linguagens de especificação que sejam acessíveis simultaneamente a projetistas, a especificadores e a programadores.

A abordagem Larch está orientada à especificação de módulos de programas a serem implementados em uma linguagem de programação particular. Cada especificação Larch tem dois componentes: uma escrita em uma linguagem derivada de uma linguagem de programação, chamada *Linguagem de Interface Larch*; e outra escrita em uma linguagem comum e independente de qualquer linguagem de programação, chamada *Linguagem Compartilhada Larch*.

Abstrações são formuladas na linguagem Compartilhada. As linguagens de Interface (orientadas a predicados) são usadas para descrever o comportamento de procedimentos. As descrições dadas nas linguagens de interface são dadas em termos destas abstrações e podem também incluir manipulações de erros, situações de exceção e limites de implementação.

Este trabalho apresenta um estudo da familia de linguagens Larch e uma aplicação das mesmas a um problema práctico. Na primeira parte, faz-se uma descrição da Linguagem Compartilhada Larch (sintaxe, semântica, consistência, completeza, gramática, núcleo) e das

linguagens de Interface Larch com exemplos orientados à lingugem de programação Pascal. Na segunda parte, apresenta-se a especificação do modelo relacional para banco de dados, também orientado a programadores em Pascal. O núcleo desta especificação está contida em [GUT 85] e é um conjunto de módulos de relações e estruturas matemáticas. Depois segue a especificação de conjunto, lista, tupla, relação e banco de dados relacional. Cada especificação, além das duas componentes já mencionadas tem uma parte de comentários que é utilizada somente para fins didáticos e entender melhor a especificação.

ABSTRACT

Recent research on the specification area is emphatic on the practical use of formal specifications in programs design. One way to satisfy this, is the supply of specification languages that could be accessible simultaneously to designers, specifiers and programmers.

The Larch approach is geared towards specifying program modules to be implemented in particular programming languages. Each Larch specification has two components: one written in a language derived from a programming language, called *Larch Interface Language*; and another component written in a language independent of any programming language, called *Larch Shared Language*.

Abstractions are formulated in the Shared Language. The Interface Language (predicate-oriented) is used to describe the intended behaviour of procedures. Descriptions given in the interface languages are given in terms of those abstractions and might also include error and exception handling situations and implementation limits.

This work presents a study of the family of Larch Languages and their aplication to a practical problem. The first part is a description of Larch Shared Language (syntax, semantics, consistency, completeness, grammar, kernel) and Larch Interface Language with examples oriented to Pascal programming language. The second part presents

the specification of the relational model for databse, also oriented to Pascal programmers. The kernel of this specification is taken from [GUT 85] and is a modules set of relations and mathematical structures. This is followed by specification of concept of set, list, tuple, relation and relational database. Each specification, besides the two component mentioned above, has a commentary used only for didactic purpose and to explain better the specification.

1. INTRODUÇÃO

Os últimos anos da década dos 70 e toda a década dos 80 tem sido caracterizada pela abundante pesquisa na área de especificação formal. Muitos métodos, modelos, linguagens e ferramentas foram desenvolvidos, primeiramente com alcance somente acadêmico e logo projetado a problemas de aplicação. Porém, hoje torna-se uma necessidade a divulgação, extensão e transformação dos métodos já existentes.

A característica principal da maioría destes métodos está no seus fundamentos matemáticos e no conceito de Tipo Abstrato de Dado. Por tal motivo, recebem o nome de *métodos algébricos* e as especificações que usam tais formalismos chamam-se *especificações algébricas*.

Os pioneiros das especificações algébricas são Zilles (1974), Goguen (1974) e Guttag (1975). Todos eles iniciaram com uma abordagem baseada na mesma estrutura: uma álgebra. A primeira linguagem de especificação algébrica foi CLEAR que apareceu em 1977. Depois outras linguagens ficaram conhecidas: ACT ONE, OBJ, SPECIAL, OBSCURE, Ina Jo, ABEL, Larch, etc. Conceitos como: Tipos Abstratos de Dados, especificação parametrizada, correção, completeza, provador de teoremas, manipulação de erros, etc. são analisados cada vez mais com maior interesse.

A escolha das linguagens Larch como objeto de estudo foi feita tomando em conta, primeiramente, a simplicidade (um único tipo de módulo, poucas cláusulas,

poucas palavras chaves) de cada especificação, que permitirá uma fácil compreensão e uma correta implementação do que está se especificando. Segundo, a familia de linguagens Larch é o resultado da experiência com outros métodos e linguagens de especificação, e portanto, é uma das mais novas e atualizadas na sua categoría. Terceiro, as linguagens Larch somente são linguagens de especificação e não de programação e podem ser utilizadas tanto para especificação de programas sequênciais como para processos concorrentes.

Como primeira contribuição deste texto, pretende-se apresentar uma descrição detalhada da familia de linguagens Larch que poderá ser utilizada como material auxiliar nas disciplinas relacionadas à Especificação Formal. Um segundo aspecto importante é a apresentação de um exemplo prático do uso das linguagens: a especificação do modelo relacional de banco de dados.

O texto está dividido em quatro partes principais. Neste Capítulo, apresentam-se os objetivos que motivaram o Projeto Larch e as características das linguagens Larch. São mencionados o processo incremental de construção das especificações, as dependências de uma linguagem de programação, entre outras. Finalizando este capítulo apresenta-se um estudo simplificado de quatro linguagens algebricas que servem para fazer uma comparação com as linguagens Larch. No Capítulo 2, são mostradas a sintaxe, semântica, consistência , completeza, gramática e núcleo da Lingugem Larch Compartilhada. A segunda parte deste capítulo é a descrição das linguagens de Interface Larch, utilizando um exemplo conhecido (pilha) em Larch/Pascal. Na parte final mostra-se uma comparação das linguagens Larch com outras linguagens de características semelhantes.

O Capítulo 3 é a especificação do Modelo Relacional para Banco de Dados usando Larch. Cada especificação (conjunto, lista, tupla, relação e banco de dados) tem três partes: especificação em Larch Compartilhada, comentarios (ou especificações informais) e especificação em Larch/Pascal. A escolha da linguagem Larch/Pascal como linguagem de interface, tem a ver com a familiaridade e uso generalizado da linguagem Pascal nos diferentes Cursos de Ciência da Computação.

O Anexo apresenta um conjunto de definições básicas relacionadas à especificação formal, para auxiliar o leitor a entender alguns conceitos que aparecem principalmente na primeira parte do Capítulo 2.

As referências bibliográficas apresentadas estão divididas implicitamente em duas categorias: uma relacionada com a familia de linguagens Larch e outra com banco de dados relacionais. Uma referência bibliográfica que serve de pre-requisito para entender alguns conceitos básicos sobre especificação algébrica, mencionados no texto, é [CAS 88].

1.1. Objetivos

O Projeto Larch do MIT-LCS e DEC's - SRC, liderado pelos professores J.Guttag e J.J.Horning, compreende a definição de uma familia de linguagens de especificação e a implementação de um conjunto de ferramentas para auxiliar sua utilização. Apresentam-se algumas das razões e problemas que orientaram e forçaram o desenvolvimento das linguagens Larch:

- *Especificações Locais.*

No inicio, o Projeto Larch foi dirigido para especificações orientadas a linguagens de programação, considerando que o uso de especificações algébricas é voltado atualmente a pequenas unidades de programas.

- *Programas Sequênciais.*

A maioría das unidades de programas são desenvolvidas em ambientes não concorrentes. Isto motivou que Larch fosse orientado para especificação de programas sequênciais. No entanto, a necessidade cada vez maior de concorrência possibilitaram o trabalho posterior de J.Guttag e J. Wing ([BIR 87], [HER 86]), como uma extensão de Larch Sequencial para processos concorrentes.

- *Especificação vs. Programa*

Com o desenvolvimento de diferentes métodos e linguagens para especificação formal, especificamente algébricos, e os resultados positivos obtidos até inicios dos anos 80, muitos sistemas passaram a incluir outros conceitos, com a finalidade de que o método e/ou a linguagem fosse mais útil (execução, prototipação, etc.). O resultado foi um conjunto de linguagens de especificação e programação ao mesmo tempo. Porém, ainda falta, nestas linguagens, uma distinção sintática e semântica, muito clara, entre a especificação de propriedades de abstrações básicas e especificação de componentes de programas.

- *Escala.*

Métodos que são adequados para especificações curtas (1 ou 2 páginas) podem ter problemas com especificações de 100 ou mais páginas. É fundamental que as "grandes" especificações sejam compostas a partir de outras

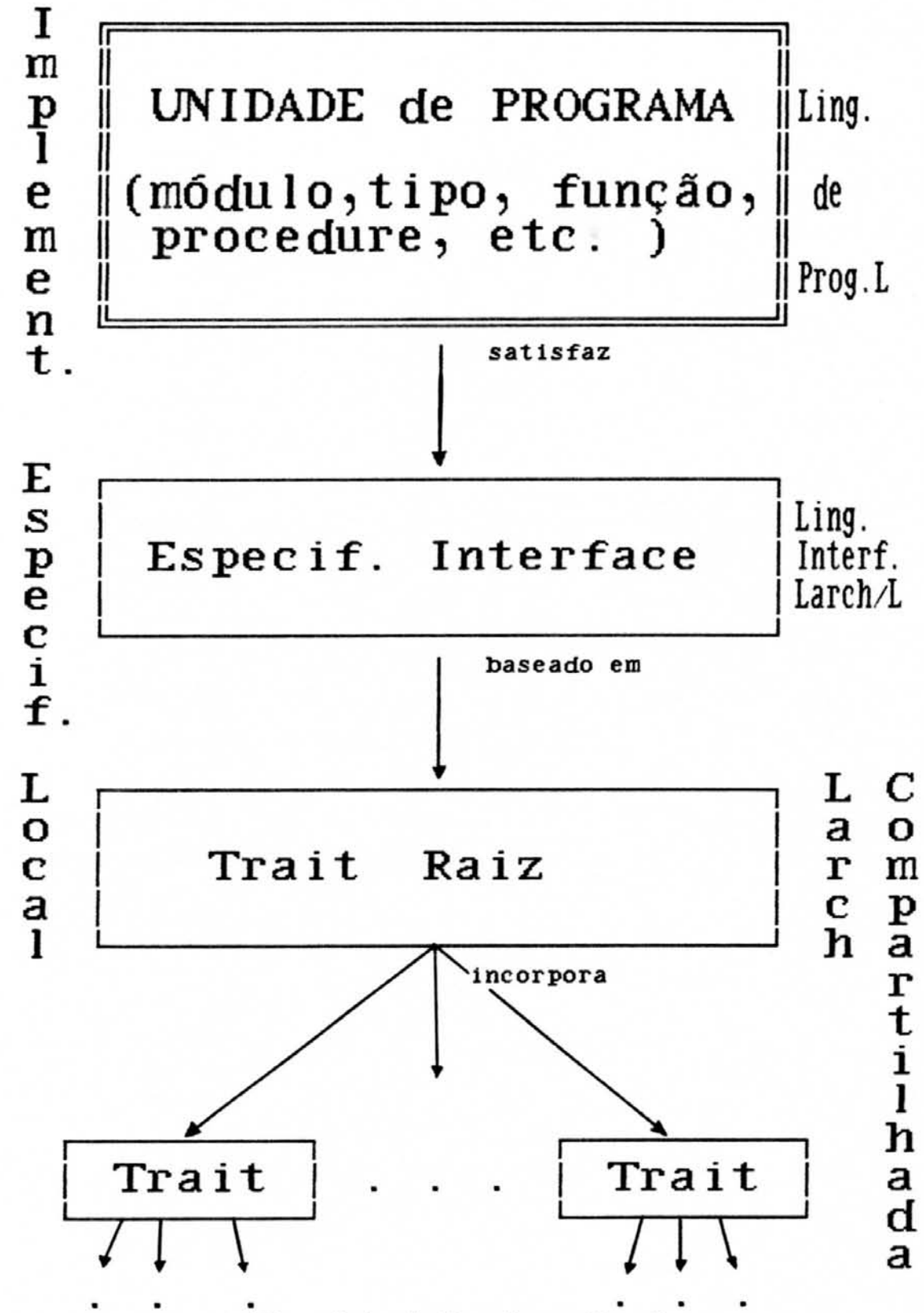

Fig. 1.1 A Abordagem Larch

"pequenas" e que estas possam ser entendidas separadamente.

- *Incompleteza.*

Na prática, todas as especificações são parciais. Não está definido o limite entre especificação e implementação. Não existe uniformidade sobre a abstração de detalhes em uma especificação, pois esta depende dos propósitos e necessidades do especificador. As vezes, tempo, uso de memoria ou funcionalidade são tomados em conta na especificação e isto depende da escolha intencional no momento de tomar a decisão. Outras vezes, certos detalhes simbolizam uma superposição no processo de especificação que deveriam ser reconsiderados. É preciso ter um método que permita detectar a classe de incompleteza que está presente na especificação.

- *Erros.*

O processo de especificação é menos propenso a erros que o processo de programação. É muito mais fácil e cômodo verificar uma especificação duas ou três vezes, que corrigir um erro de um programa implementado em alguma linguagem de programação. Uma linguagem de especificação que permita verificação redundante,manual ou mecânicamente, é necessária.

- *Ferramentas.*

Um dos grandes problemas de usar especificações formais é a quantidade de tarefas relacionadas a evitar erros e a manter a consistência do texto da especificação. Ferramentas que ajudem a gerenciar a relação entre grandes e pequenos módulos de especificações, que permitam interação usuário-especificação são muito

importantes em uma linguagem de especificação formal.

- *Bibliotecas e Reusabilidade.*

É muito ineficiente desenvolver uma especificação desde o inicio. O conceito de reusabilidade implica a existência de um conjunto de componentes de especificação reusáveis, a serem facilmente aproveitados em casos comuns e que sirvam de modelos para casos não comuns. É necessário pacotes de módulos de especificação básicos e outros orientados a aplicações.

1.2. Características

Entre as principais características da familia de linguagens Larch tem-se:

a) **COMPOSIÇAO.**

Cada especificação em Larch, geralmente é composta de outras especificações, de modo que a construção é incremental: uma a partir de outras. Esta é a maior vantagem em relação as outras linguagens de especificação algébrica.

b) **ENFASE NA APRESENTAÇAO.**

A linguagem Larch foi projetada para construir especificações que possam ter uma leitura FACIL, tanto pelo especificador como pelo usuário implementador. Os mecanismos de composição em Larch (inclusão, importação,etc.) são definidos como operações sobre especificações, em vez de

teorias ou modelos.

c) **APROPRIADA PARA FERRAMENTAS INTEGRADAS E INTERATIVAS.**

As linguagens Larch foram projetadas pensando na implementação de ferramentas que permitam a construção interativa e a verificação incremental de especificações.

d) **VERIFICAÇAO DA SEMANTICA.**

A medida que as especificações estão sendo construídas, as linguagens Larch habilitam uma verificação extensiva. Faz parte do Projeto Larch a utilização de um poderoso Provador de Teoremas para a verificação semântica de especificações, complementando a verificação sintática geralmente definida para linguagens de especificação.

e) **DEPENDENCIAS DA LINGUAGEM DE PROGRAMAÇAO LOCALIZADAS.**

Cada linguagem de Interface Larch encapsula as características necessárias para escrever especificações concisas e compreensíveis para uma linguagem de programação particular (Pascal, C, CLU, Ada, etc.) e incorpora uniformemente todas as especificações escritas em Larch Compartilhada.

f) **REUSABILIDADE.**

Componentes de Larch Compartilhada podem ser re-usadas por componentes de linguagens de interface diferentes.

g) **LINGUAGEM COMPARTILHADA BASEADA EM EQUAÇOES.**

A linguagem Compartilhada possui uma base semântica simples extraída da álgebra. A ênfase na

sua composição, verificabilidade e interação, faz a diferença com outras linguagens algébricas.

h) **LINGUAGENS DE INTERFACE BASEADAS EM CALCULO DE PREDICADOS.**

Cada linguagem de interface esta baseada em asserções escritas em cálculo de predicados de primeira ordem tipado, com igualdade, e incorpora fatos específicos da linguagem de programação, que permita manipular situações excepcionais, erros, etc.

1.3. Ferramentas

O Projeto Larch tem desenvolvido um conjunto de ferramentas para auxiliar a construção de especificações formais em Larch. Estas são:

a). **PISTOL**: É um verificador sintático e semântico (estático) para Larch Compartilhada e é descrito em [KOW 84]. No estilo de um compilador, Pistol analisa especificações e somete elas a verificações sensíveis ao contexto (Ex. variáveis não declaradas, variáveis duplicadas, etc.). Sua enfase é nos problemas e reportagens destes para o usuário. As verificações são executadas em camadas com as ocorrências de erros livres do contexto eliminando a consideração de verificações sensíveis ao contexto ([ZAC 83]).

b). **SDSE**: é um Editor dirigido por sintaxe para Larch Compartilhada, e descrito em [ZAC 83]. O editor (abstrato) é considerado como uma máquina abstrata que

opera sobre objetos abstratos (especificações). O editor concreto (interface com o usuário) fornece ao especificador os mecanismos para chamar as operações do editor abstrato. Um mapeamento de objetos para textos representados visualmente, permite apresentar textos da especificação que está sendo desenvolvida junto com os erros sensíveis ao contexto.

c). Um verificador semântico (baseado no poderoso sistema REVE, um provador de teoremas) que pode manipular equações é descrito em [FOR 85].

d). Uma biblioteca de especificações usando Larch Compartilhada é apresentada em [ATR 82] e contida na parte IV de [GUT 85].

e). O Provador Larch, LP: pode ler uma descrição de uma teoría equacional e tentar provar sentenças nesta teoría (por re-escrita de termos). Uma descrição geral encontra-se em [GAR 88].

f). O Verificador Larch, LC: lê um trait (o modulo principal da linguagem Larch Compartilhada) e escreve dois arquivos como entrada para o LP: uma axiomatização do trait e um texto de comandos para descarga de obrigações de prova associadas a um trait. É explicado em [GAR 89].

g). Penelope: é um editor para auxiliar a programadores a desenvolver e verificar programas especificados em Larch/Ada-88. Penelope foi implementado no DCS-Princeton University, usando o Cornell Synthesizer Generator, e tem três componentes: de transformação de predicados, de prova e de simplificação. Penelope é

descrito em [RAM 89].

O autor, somente dispõe da especificação formal da segunda ferramenta acima mencionada.

1.4 Linguagens de Especificação Algébrica

A seguir apresenta-se um resumo e exemplos de quatro linguagens de especificação algébrica: ABEL, OBJ, ASL e Iota, que serão usadas para fazer uma comparação com a linguagem Larch (Compartilhada). Outras linguagens (ou sistemas) algébricos como Ina Jo, ACT-ONE, SPECIAL, OBSCURE e SEKI, não foram consideradas pela falta de suficientes ou atualizadas referências bibliográficas.

1.4.1. ABEL (Abstraction Building Experimental Language)

A B E L (University of Oslo) ([DAH 86]) é uma linguagem (Larch-like, [DAH 89]), para especificação formal e programação. Foi desenvolvida inicialmente como um auxilio para o ensino de técnicas de especificação formal e raciocínio auxiliado por máquina, com programação imperativa e verificação de programas como casos especiais.

A linguagem contém:

- mecanismos para especificações construtivas e não-construtivas
- programação imperativa e aplicativa.

O projeto **ABEL** teve seu inicio em 1976, com versões em 1977,

1978,1984 e 1986. As ultimas versões de **ABEL** foram inspiradas na linguagem **LARCH**. Atualmente, outras duas linguagens foram incluidas no projeto:

- **BABEL**, para projeto e implementação de linguagens, e
- **CABEL**, orientada a processos concorrentes.

A estrutura principal de **ABEL** é o modulo ("module"), construído por encapsulamento de funções e possívelmente tipos. As especializações de um módulo são:

-**type** module : define um ou mais tipos e funções associadas .
-**class** module : para classes Simula-like.
-**group** module : define grupos de funções em um tipo.
-**property** module : especifica requerimentos mínimos de tipos e funções.

Tipos são definidos por meios algébricos ou por referências a tipos previamente definidos.

Um *subtipo* herda o conjunto de valores e o conjunto de funções associadas a seu respetivo supertipo, e este pode restringir subtipo, agregar novas funções e redefinir outras.

As funções podem ser definidas:

- não-construtivamente (conjunto de axiomas de primeira ordem).
- construtivamente (definições equacionais, ver [DAH 89])

A semântica de todos os mecanismos de definição aplicativos, são definidos agregando axiomas e regras de inferência a um sistema formal básico de lógica de primeira ordem multi-sortida.

A linguagem é orientada a objetos (não existe aninhamento de módulos e definições de funções, [DAH 88])

A linguagem **ABEL** tem duas construções imperativas principais:

- seção **prog** (com aatribuições e operações I/O).
- **class modules** , que produzem "valores": os objetos no sentido imperativo convencional.

A linguagem **ABEL** possui 82 palavras reservadas.

Exemplo 1.

```
type FinSet(T) ==
module
     func null =: FinSet                    -- empty set
          FinSet add T =: FinSet            -- add an element
          FinSet has T =: Boolean           -- membership test
          FinSet del T =: FinSet            -- delete an element
     genbas null, 'add'
     def s has x == case s of null --> f|| s add y -->
                              x = y v s has x fo
     def s del x == case s of null --> null || t add y -->
         if x = y then t del x else t del x add y fi fo
     obsbas 'has'
     lemmas ( st:FinSet, x,y:T )
          s add x add x = s add x
          s add x add y = s add y add x
     endmodule
```

Exemplo 2.

```
property LinOrd(T) assuming TotOrdAug(T) ==
module
     T func ST =: T
            PT =: T
```

```
axioms (x,y:T)
        Px < x < Sx
        PSx = x = SPx
        ( x < y ) = ( Sx ≤ y ) = ( x ≤Py )
endmodule
```

1.4.2. OBJ ([GOG 84], [GOG 86], [DUC 87])

OBJ é uma linguagem formal para escrever e testar especificações algébricas. Pode ser considerada também como uma linguagem de programação, pois as especificações escritas em OBJ podem ser executadas.

OBJ usa equações para definir Tipos de Dados Abstratos.

Uma especificação OBJ tem uma estrutura modular com componentes básicos chamados *objetos*. A sintaxe para objetos foi inspirada na notação para álgebras iniciais multi-sortidas, i.e., um objeto denota uma álgebra inicial multi-sortida particular.

Cada especificação OBJ tem:

- Semântica denotacional algébrica, baseada em lógica operacional,
- Semântica operacional, baseada na interpretação de equações como regras de re-escrita.

Um objeto em OBJ é definido iniciando pela palavra chave OBJ e um identificador simples e único, e está dividido em três partes:

- seção de sorts (sorts e subsorts),
- seção de operações e atributos, e
- seção de equações.

OBJ distingue três tipos de operadores:

- para situações ordinarias ou ok,
- para situações excepcionais ou erro, e
- para situações de recuperação ou fix.

Em uma especificação OBJ existem dois tipos de equações:

- para expressões ok (OK-EQNS), e
- para expressões de erro (ERR-EQNS).

A linguagem possui três objetos incorporados: INT (para inteiros), BOOL (para booleanos) e ID (para identificadores).

Para satisfazer as exigências da programação parametrizada, OBJ incorpora conceitos de teoría, visão, expressões módulo e instanciação.

Uma *teoría* expressa as propriedades globais de um módulo. A diferença entre objeto e teoría é que uma teoría não é executável e o objeto sim. Uma teoría pode usar outras teorías, outros objetos, pode ser parametrizada e pode ter visões.

O propósito de uma *visão* V(M) é mostrar explicitamente, como um módulo M (objeto ou teoría) satisfaz outra teoría T (mapeamento de T em M):

```
V(M) = {   Vs : sorts(T) --------> sorts(M)
           Vo : Ops(T)   --------> Ops(M)        },
```

onde Vs preserva as relações de subsort e Vo preserva aridade, valores e atributos.

Uma *expressão módulo* é uma expressão que define (por combinação ou modificação) um novo módulo a partir de

outros módulos já existentes e de acordo com um conjunto específico de operações.

Exemplo 1: ([GOG 84])

```
OBJ BSTACK-OF-INT / INT NAT
     SORTS STACK
     SUBSORTS NE-STACK < OK-STACK < STACK
     OPS
          EMPTY : STACK
          PUSH : INT STACK -> STACK
          POP : NE-STACK -> OK-STACK
          TOP : NE-STACK -> INT
          DEPTH : OK-STACK -> NAT
     VARS
          I: INT; S: OK-STACK
     SORT-DECLS
          (AS NE-STACK: PUSH(I,S) IF DEPTH(S) < 10000 )
     EQNS
          (DEPTH(EMPTY) = 0)
          (DEPTH(PUSH(I,S) ) = INC(DEPTH(S) )  )
          (POP(PUSH(I,S)) = S)
          (TOP(PUSH(I,S)) = I)
ENDO
```

Exemplo 2: ([GOG 86])

```
OBJ COND#2
SORTS / INT BOOL
FIX-OPS
     IF_THEN_ELSE_FI : BOOL INT INT -> INT
ERR-OPS
     IF-ERR_ : BOOL -> INT
VARS
  I J : INT
  B : BOOL
```

EQNS

(IF T THEN I ELSE J FI = I)

(IF F THEN I ELSE J FI = J)

ERR-EQNS

(IF B THEN I ELSE J FI = IF-ERR B IF ERR B)

JOB

Atualmente existem muitas versões e implemetações de OBJ: OBJ-0, OBJ-1, OBJ-2, OBJ-3, OBJ-IMAGEN, OBJ-T

1.4.3. A S L (Algebraic Specification Language, [HOR 88])

Van Horebeek, da Katholieke Universiteit Leuven, Belgica , tem uma proposta de uma linguagem de especificação algébrica (sem nome definido. ASL é uma sugestão) baseada em álgebras iniciais multi-sortidas.

Como em ABEL e OBJ, a estrutura principal de ASL é o **módulo**, i.e., cada especificação é construída a partir de módulos. Sorts, operações, declarações e/ou axiomas formam um módulo. No entanto, é possível ter outra estrutura chamada **cluster** (supermódulo).

A sintaxe de uma especificação em ASL tem a forma seguinte:

< specification > = (< module >)+

<module> =

"module" [<module name>]";"

[<import clause>]

[<sorts part>]

[<operations part>]

[<declarations part>]

[<axioms part>]

"end" "module" [<module name>]";"

<import clause> =
'import' 'all' 'from' < module name list>

<module name list> = <module name> (","<module name>)*

<sorts part> =
(**'sort'** | **'sorts'**) (<sort name> ";")+

Geralmente, cada módulo contém no máximo um sort. Através da cláusula *import*, um módulo pode importar direta ou indiretamente outros módulos. Um módulo sem a cláusula *import* é chamado *módulo primitivo*.

Um Tipo de Dado Abstrato (álgebra inicial) é definido por um grafo de módulos, onde as cláusulas *import*, formam os arcos (os relacionamentos de dependência). Em geral, uma especificação é um grafo dirigido. Um caso especial, são as **especificações hierárquicas**, onde o grafo é aciclico. Um *cluster* é um pacote simples de módulos dentro de um laço em um grafo dirigido.

Algumas construções que fazem parte da linguagem ASL são:

- ifthenelse.
- case.
- let ... in ...
- rename ... as ...

ASL permite especificações parametrizadas através dos mecanismos:

- **schema** (funções), e
- **instantiate** (instanciação).

Exemplo 1: ([HOR 88], p. 92,93)

```
scheme StackScheme [
     requeriment Item;
          export all;
          sort Item;
          operation
             error: -> Item;
     end requeriment Item;
     ];

  module Stack;
     import Bool, true,false from Bool;
        all from Item;
     export all;
     sort Stack;
     operations
          newstack: -> Stack;
          push: Stack * item -> Stack;
          isnewstack: Stack -> Bool;
          pop: Stack -> Stack;
          top: Stack -> Item;
     declare s:Stack; i:Item;
     axioms
       isnewstack( newstack )  ==  true;
       isnewstack( push(s,i) ) == false;
       pop( newstack ) == newstack;
       pop( push(s,i)) == s;
       top( newstack ) == error;
       top( push(s,i) ) == i;
  end module Stack;
end scheme StackScheme;
```

```
instantiate StackScheme;
     with Item as Nat,
          error as zero;
end instantiate StackScheme;
```

Exemplo 2: (Extraído e simplificado de [HOR 88], p 116,117)

```
module Position;
     import
     export Pos, forward to Commands; ...
     sort Pos;
     operations
          makePosition : ... -> Pos;
          forward: Pos -> Pos;
          . . .
end module Position;


cluster Robot;
 module Library;
     import Com from Commands;
            iden from Identifiers;
     export Lib to Enviroment, Commands;
     sort Lib;
     .........
 end module Library;

 module Enviroment;
     import Lib from Library;
     export .....
     ............
 end module Enviroment;
```

```
module Commands;
    import Env, makeEnv from Enviroment;
           Pos, forward from Position;
           . . . . . .
end module Commands;

end cluster Robot;
```

Exemplo 3: (Extraído de [HOR 88], p. 227,228)

```
module BoundedStack;
  import Bool, true, false, errBool from Bool;
    Nat, zero, succ, _≤_, errNat from Nat;
  export all ;
  sort BoundedStack;
  constructors
   newstack : -> BoundedStack $$;
   push : $ BoundedStack * Nat -> BoundedStack
              $ length ( boundedstack ) ≤ 99 $;
   underflow : -> BoundedStack ??;
   overflow : -> BoundedStack ??;
  operations
   length : BoundedStack -> Nat;
   pop : BoundedStack -> BoundedStack;
   top : BoundedStack -> Nat;
   isnewstack : BoundedStack -> Bool;
   recover : BoundedStack -> BoundedStack;
  declare b: BoundedStack; n: Nat;
  constructor axioms
   !push( $ b, n ) == overflow;
    push( ! b, n ) == b;
```

```
operation axioms
  $$ length( newstack ) == zero;
  $$ length( push(b,n) ) == succ( length(b) );
  $$ pop( newstack ) == underflow;
  $$ pop( push(b,n) ) == b;
  $$ top( newstack ) == errNat;
  $$ top( push(b,n) ) == n;
  $$ isnewstack( newstack ) == true;
  $$ isnewstack( push(b,n) ) == false;
  $$ recover( b ) == b;
  ?? length( b ) == errNat;
  ?? pop( b ) == b;
  ?? top( b ) == errNat;
  ?? isnewstack( b ) == errBool;
  ?? recover( b ) == newstack;
end module BoundedStack;
```

Entre alguns casos de estudos onde já foi utilizada esta linguagem, temos: a especificação industrial de um Sistema de Manipulação de Chamadas (mini-PABX), especificação formal de um Sistema de Robot Pequeno (Karel The Robot), especificação do problema do bote, especificação de um sistema de arquivos UNIX-like, especificação de um pacote gráfico incluindo GKS, e outras descritas em [HOR 88].

A linguagem ASL, possui já algumas ferramentas como: editor dirigido pela sintaxe, um compilador, um provador de teoremas, e outras.

1.4.4. Iota ([NAK 83])

Iota é uma linguagem de especificação e

programação modular, que foi desenvolvida no IMS da Universidade de Kyoto, Japão.

Iota usa uma lógica de primeira ordem multi-sortida chamada *iota-lógica*, incluindo a regra de indução (gerador) sobre um sort ([DAL 89]). A estrutura básica de um sort S em iota-lógica está formada por:

- um conjunto finito de funções PR<S> chamado *funções primitivas*, e
- um conjunto finito de axiomas BA<S> chamado *axiomas básicos*.

A noção de uma especificação em **Iota** consiste de uma hierarquia (árvore) de componentes modulares chamados *"módulos"*, cada um dos quais define uma abstração de dado ou procedural ([LIS 86]). Existem basicamente três classes de módulos: type, sype e procedure. A diferença entre type e sype é a não existência da noção de regra de indução sobre sypes.

Exemplo 1.

```
interface type NN
     fn ZERO: -> 0 as 0
        SUC:  0 -> 0
        LESS: (0,0) -> BOOL as 0 ≤ 0
      [EQUAL: (0,0) -> BOOL as 0 = 0 ]
end interface
specification type NN
var X,Y,Z : 0
axiom 1: X = Y ⊂ SUC(X) = SUC(Y)
      2: ~SUC(X) ≤ X
      3: SUC(X) ≤ SUC(Y) ⊂ X ≤ Y
      4: X ≤ Y  ⩛ Y ≤ X
      5: X=Y ⊂ X≤Y ∧  Y≤X
      6: X≤Z ⊂ X≤Y ∧ Y≤Z
end specification
```

Exemplo 2.

```
interface sype ORDER
      fn  LESS: (0,0) -> BOOL as 0 ≤ 0
        [EQUAL: (0,0) -> BOOL as 0 = 0 ]
end interface
specification sype ORDER
      var X,Y,Z : 0
      axiom 1: X ≤ Y  ∨ Y ≤ X
            2: X≤Z ⊂ X≤Y ∧  Y≤Z
            3: X=Z ⊂ X≤Y ∧ Y≤X
end specification
```

Exemplo 3.

```
interface procedure INTSEARCH
      fn SORTED:  INTARRAY -> BOOL
         LOCATE:  (INTARRAY,INT) -> (BOOL,NN)
end interface
specification procedure INTSEARCH
      var X:  INTARRAY; M,N:  NN; I:  INT
      axiom 1:  SORTED(X) ≡ ∀M, .......
                ..........................
end specification
```

Exemplo 4.

```
realization type INTPOLY
      rep = INTARRAY
      fn |COEF(X:rep, N:NN) return (  I:INT)
          if N ≤ HIGH(X)
            then I:= X[N]
            else I:= 0  end if
      end fn
      fn |ZERO return (X:rep)
           X:= CREATE(0,0)
      end fn
      ..................
end realization
```

Formalmente, cada módulo m define uma teoría TH(m) em iota-lógica (sorts, funções e fórmulas). **Iota** possui mecanismos de implementação ("realization") e parametrização de types e sypes.

Tendo em conta que, um módulo m é especificado e implementado, a teoría TH(m) deve ser provada para satisfazer a parte "realization". O sistema **Iota** esta constituído de cinco grandes subsistemas: Ambiente de criação e modificação de módulos (Editores e Analisadores), Depurador, Verificador, Provador (de fórmulas) e Executor (Tradutor e Carregador).

2. AS LINGUAGENS LARCH

2.1 Linguagem Larch Compartilhada (LLC)

O objetivo desta parte é apresentar detalhadamente a linguagem de especificacão Larch Compartilhada. Cada componente da sintaxe e semântica da linguagem LC é precedida por uma porção de gramática que corresponde à parte que está sendo descrita, bem como uma rápida descrição do processo de verificação.

A maior parte das convenções sintaticas que são usadas neste capítulo são apresentadas na seção 2.1.3.1 .

2.1.1. Gramática da Linguagem Larch Compartilhada

```
      trait ::= traitId : trait traitBody finals
  traitBody ::= externals simpleTrait
  externals ::= {assumes} {imports} {includes}
    assumes ::= assumes traitRef*,
    imports ::= imports traitRef*,
   includes ::= includes traitRef*,
   traitRef ::= traitId {renaming}
   renaming ::= with [(sortRename | opRename)*,]
 sortRename ::= sortId for oldSort
    oldSort ::= sortId
   opRename ::= opId for oldOp
      oldOp ::= sortedOp
   sortedOp ::= signature
```

```
simpleTrait ::= {opPart} propPart*
     opPart ::= introduces signature+
  signature ::= opId : domain --------> range
     domain ::= sortId*,
      range ::= sortId
   propPart ::= (asserts | constrains) props
 constrains ::= constrains (sortId | sortedOp*, ) so that
      props ::= generators* partitions* axioms*
 generators ::= sortId generated bylist*,
 partitions ::= sortId partitioned bylist*,
     bylist ::= by [ sortedOp*, ]
     axioms ::= for all [ varDcl*;  ] equation*
     varDcl ::= varId*, : sortId
   equation ::= term { =term }
       term ::= sec | if sec then sec else term
        sec ::= {opSym} prim (opSym prim)*  {opSym}
       prim ::= sortedOp { '(term*,') } | varId | '(term ')
       opId ::= alphaNumeric+ | opForm
     opForm ::= {#} opSym (# opSym)* {#}
      opSym ::= specialChar+ | .alphaNumeric+
    traitId ::= alphaNumeric+
     sortId ::= alphaNumeric+
      varId ::= alphaNumeric+
     finals ::= {consequences} {exempts}
consequences::= implies conseqProps {converts}
conseqProps ::= traitRef*, props
   converts ::= converts conversion*,
 conversion ::= [ sortedOp*, ]
    exempts ::= exempts exemptTerms*
exemptTerms ::= { for all [ varDcl*; ] } term*
```

2.1.2 Núcleo da Linguagem Larch Compartilhada

O núcleo da LLC é um conjunto de traits (módulos escritos em LLC e que são estudados na seção seguinte) que servem principalmente para três propósitos:

- fornecer um conjunto de exemplos demonstrativos para auxiliar aos usuários a compreender a linguagem Larch Compartilhada.
- fornecer um conjunto de componentes que possam ser diretamente incorporados em outras especificações.
- fornecer um conjunto de modelos a ser utilizados na modelagem de outras especificações, visto que alguns especificadores preferem EDITAR uma especificação a INCLUIR outra já pronta.

Os traits que integram o núcleo da LLC estão agrupados na seguinte forma:

- Propriedades básicas de operadores simples.
- Propriedades básicas de relações binarias.
- Ordem em relações (parcial e total).
- Teoría de grupos (identidades, semigrupos, monoides, grupos,etc.).
- Tipos numéricos simples (ordinal, cardinal).
- Estrutura de dados simples (par, triple, etc.).
- Propriedades de conjunto.
- Classes de conjunto (conjunto, pilha, fila, sequência) .
- Operadores genéricos sobre conjuntos.
- Estruturas não lineares (arvores, grafos).
- Anéis, corpos e números.
- Reticulados (lattice).
- Tipos de dados enumerados.
- Traits elementares para Computação Gráfica.

O conjunto de traits integrando o Núcleo da LLC está

contido na parte IV de [GUT 85].

2.1.3 Sintaxe e Semântica de uma Especificação

Definição. TRAIT

Um *trait* (ou tipo) é a unidade básica de especificação na Linguagem Larch Compartilhada (LLC). Um trait:

- introduz *operadores*, e
- especifica as *propriedades* destes operadores.

Algumas vezes, um trait (a coleção de operadores e o conjunto de propriedades) corresponderá a um Tipo de Dado Abstrato (TDA). Em geral, traits são considerados como simples objetos textuais.

2.1.3.1. Sintaxe

CONVENÇÕES SINTÁTICAS:

|	:	separador alternativo
{ expr }	:	expr é opcional
expr*	:	zero ou mais expr's
expr*,	:	zero ou mais expr's separadas por vírgulas
expr*;	:	zero ou mais expr's separadas por ponto e
	:	vírgulas.
expr+	:	uma ou mais expr's
'(')	:	parêntesis como símbolos terminais
(expr)	:	parêntesis para agrupar expresões sintáticas
%	:	inicio e fim de comentários
negrito	:	símbolo terminal
sortId	:	identificador de sort
opId	:	identificador de operador

Gramática:

```
        trait ::= traitId : trait traitBody  finals
    traitBody ::= { externals } simpleTrait
  simpleTrait ::= { opPart } propPart*
       opPart ::= introduces signature*
    signature ::= opId : domain --------> range
       domain ::= sortId*,
        range ::= sortId
     propPart ::= asserts | constrains generators*
                                        partitions* axioms*
       finals ::= { consequences } { exempts }
```

Verificação :

- Os conjuntos de identificadores de sort (**sortIds**) e identificadores de operadores (**opIds**) aparecendo em um trait (**simpleTrait**) devem ser disjuntos.
- Cada identificador de sort (**sortId**) e cada identificador de operador (**opId**) aparecendo em qualquer parte de um trait simples, i.e., sem importação ou inclusão de outros traits (**simpleTrait**), deve aparecer na parte introdutoria do trait: entre **introduces** e **constrains** (**opPart**).

Definição. A sintaxe de uma especificação na LLC é mostrada na fig. 2.1 (página seguinte) :

traitId é o nome que identifica o trait e é usado na verificação de outras especificações no momento da importação do trait *traitId*. Este nome não tem nenhuma relação lógica com os nomes que aparecem dentro do trait. Por exemplo, TABLESPEC é o identificador para um trait que especifica uma classe de tabelas (o sort Table) que armazena valores em lugares indexados, e que poderia ser renomeado para TABLEINDEX ou INDEXTABLE sem alterar o significado dos

```
traitId : trait
        { cláusulas para referências externas }
{ introduces
        { operadores }
  (constrains | asserts)* ( sortId | sortedOp*,)
  so that { sortId generated by [ sortedOp*, ]
        { sortId partitioned by [ sortedOp*,] } }
     for all [( varId*, : sorteId)*, ] { axiomas }
       { cláusulas para consequencias }
```

Fig. 2.1 Sintaxe de uma Especificação em LLC

operadores ou axiomas que aparecem no trait. Neste texto, o identificador de um trait (traitId) sempre será escrito em letras maiúsculas.

A sintaxe de um trait simples (sem referências externas) é definida pela cláusula

```
introduces
     op1 : domain1 ----------> sortId1
     . . . . . . . . . . . . . . . . . .
     opn : domainn ----------> sortIdn
```

A palavra chave "**introduces**" declara um conjunto de operadores (identificadores) do trait, cada um com sua respetiva signatura < { opId }, {s_dom, s_imag} > ou:

```
opId : s_dom --------> s_range
```

As signaturas são usadas depois para verificar os sorts que aparecem nos axiomas do trait. Neste texto, os identificadores de operador (opId) serão escritos em letras minúsculas, e os identificadores de sort, somente a primeira letra será escrita em letra maiúscula.

Exemplo:

BAG é identificador de trait.

bag é identificador de operador.

Bag é identificador de sort.

Na linguagem Larch Compartilhada, utilizam-se as palavras *operador*, *sort* e *termo* para evitar alguma confusão com conceitos similares de função, tipo e expressão das linguagens de programação, respectivamente.

Exemplo. (Primeira parte do trait TABLESPEC, fig 2.2)

```
TABLESPEC : trait
    introduces
        new : {}                    ---> Table
        add : Table, Index, Val     ---> Table
    element : Table, Index          ---> Bool
       eval : Table, Index          ---> Val
    isempty : Table                 ---> Bool
       size : Table                 ---> Card
```

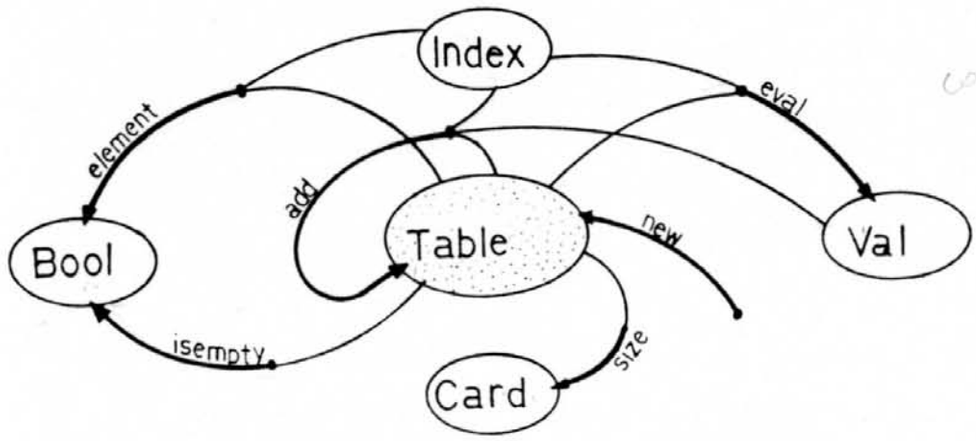

Fig.2.2 O trait TABLESPEC (1a.parte)

Observe neste exemplo que:

TABLESPEC é o identificador do trait.

{new, add, element, eval, isempty, size } é o conjunto de operadores introduzidos pelo trait TABLESPEC.

⟨Table, Index, Val, Bool, Card⟩ é o conjunto de sorts que são utilizados pelos operadores do trait TABLESPEC.

2.1.3.2 Semântica

Gramática :

```
   propPart ::= (asserts | constrains) props
 constrains ::= constrains (sortId | sortedOp*, ) so that
      props ::= generators* partitions* axioms*
     axioms ::= for all [ varDcl*; ] equation*
     varDcl ::= varId*, : sortId
   equation ::= term { =term }
       term ::= sec | if sec then sec else term
        sec ::= ⟨opSym⟩ prim (opSym prim)*  ⟨opSym⟩
```

Verificação :

- Cada identificador de variável (**varId**) usado em um termo (**term**) deve aparecer em exatamente UMA declaração de variável (**varDcl**).
- Nenhum identificador de variável (**varId**) pode ocorrer mais de uma vez na parte [**varDcl***,].
- Numa equation, os sorts de ambos termos devem ser os mesmos, onde:
  - O sort de um termo da forma *sortedOp ('(term*,')')* é a imagem de *sortedOp*.
    Exemplo:
    t1 = left(t1,d,tr) é a imagem de left .
  - O sort de um termo da forma *varId* é o identificador de sort da declaração de variáveis (**varDcl**) na qual *varId* é declarado.
- Em termos da forma *sortedOp ('(term*,')')* o domínio de *sortedOp* deve ser a sequência dos domínios

(sorts) dos termos em *term**,. No exemplo anterior, o domínio de **left** é a sequência:

dom(t1), dom(d), dom(tr)

A semântica de uma especificação em LLC corresponde à teoría associada a um trait (definida na parte B de 2.1.3.2).

Observações:

a) Nos próximos exemplos utilizam-se as cláusulas **'assumes...'**,**'imports...'** e **'includes ...'**, para referenciar outros traits que supõe-se previamente definidos. A inclusão de qualquer destas três cláusulas em um trait T, significa a inclusão em T dos sorts, operadores e axiomas definidos nos traits que aparecem em cada cláusula (assumes, imports, includes). O significado exato e as diferênças destas cláusulas posteriormente serão explicadas detalhadamente.

b) Tendo em conta a gramática da Linguagem Larch Compartilhada, um axioma tem a seguinte forma:

axiom ::= **for all** [varDcl*;] term (=term)

onde

term

é chamado uma *inequação* (ou equação implicita: term = true)

e

term = term

é uma equação propriamente dita. Logo são axiomas:

- **for all** [x:T] isempty(new())
- **for all** [x:T] not(isempty(insert(new(),x)))
- **for all** [t1,t2:T] rel(t1,t2) = rel(t2,t1)
- **for all** [f:A; s:B] first(p(f,s)) = f
- etc.

a) Semântica de um Trait

Definição. A semântica de um trait é definida pelo conjunto de axiomas que aparecem imediatamente após a cláusula:

constrains (sortId | sortedOp*,) **so that**
for all [(varId*, : sortId)*;]

ou após a cláusula:

asserts **for all** [(varId*, : sortId)*;].

constrains .... so that é a cláusula para indicar o sort (*sortId*) ou o conjunto de operadores (*sortedOp*,)* que serão restritos imediatamente pelos axiomas do trait. A lista dos operadores restritos geralmente formam um subconjunto proprio dos operadores aparecendo nos axiomas, pois este pode incluir outros operadores definidos nas referências externas (importados ou incluídos).

Se o sort *sortId* fizer parte do domínio e/ou contradomínio de cada um dos operadores aparecendo no conjunto *sortedOp*,*; então a cláusula

constrains sortId **so that**

é equivalente a

constrains sortedOp*, **so that**

i.e., o sort *sortId* representa (substitui) o conjunto de operadores *sortedOp**, .

A cláusula **asserts** é usada quando constrains não fornecer nenhuma informação (possívelmente exista um único sort sortId).

for all representa o quantificador universal ∀ (para todo) que não é um caractere ASCII (não está disponível nos teclados alfanuméricos dos computadores nem

na maioría dos editores de texto).

[(varId,: sortId)*,]* é o campo do quantificador universal *for all* e contém as listas de variáveis (e seus respetivos sorts) que devem aparecer imediatamente, nos termos dos axiomas.

Exemplo 1. (Segunda parte do trait TABLESPEC)

```
constrains new, add, element, eval, isempty, size so that
     for all [ ind, ind1: Index; val:Val; t:Table ]
     eval(add(t,ind,val),ind1)= if(ind=ind1)
                        then val
                        else eval(t,ind1)
     element(new(),ind) = false
     element(add(t, ind1, val),ind) =
                                (ind=ind1) or element(t,ind)
     size(new()) = 0
     size(add(t, ind,val)) = if element(t,ind)
                              then size(t)
                              else size(t) + 1
     isempty(t) = ( size(t) = 0 )
```

Neste exemplo o conjunto de operadores restritos pelos axiomas é:

L1 = { new, add, element, val, isempty, size }

e o conjunto de operadores que aparecem nos axiomas é:

L2 = { new, add, element, val, isempty, size, false, or, +, ifthenelse, 0 }

Obviamente, L1 é um subconjunto proprio de L2. Além disso, observando a fig. 2.2, a cláusula

```
constrains new, add, element, eval,
                    isempty, size so that
```

pode ser substituída pela cláusula

```
constrains Table so that
```

Exemplo 2. O seguinte trait especifica a propriedade distributiva para conjuntos (fig. 2.3) :

DISTRIBUTIVE_PROPERTY : **trait**

introduces

+ : S, S --------> S

* : S, S --------> S

asserts

for all [x,y,z : S]

(x, +(y,z)) = +((x,y), *(x,z)) % *(a,b) = a*b %

(+(x,y), z) = +((x,z), *(y,z)) % +(a,b) = a+b %

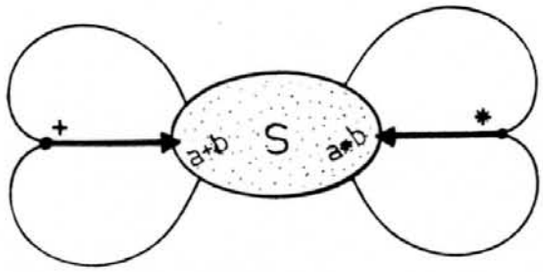

Fig. 2.3 O trait DISTRIBUTIVE_PROPERTY

No exemplo anterior, o uso da cláusula **asserts** deve-se ao único sort S. Os operadores + e * somente atuam dentro do sort S

b) Teoria Associada a um Trait

Definição. Uma TEORIA associada a um trait T, denotada por Th(T), é um conjunto de fórmulas bem-formadas (fbf's) (ver [CAS 88]) de cálculo de predicados de primeira ordem tipado com equações como fórmulas atômicas.

A teoria Th(T) para um trait T, é definida por:

i) *Axiomas*: Cada equação, universalmente quantificada

pela declaração de variáveis

for all [(varId*, : sortId)*,]

de T, está em Th(T).

ii) *Inequação*: Sejam True : {} --------> Bool

False : {} --------> Bool

A inequação **not(True = False)** está sempre em Th(T). Outras inequações em Th(T) são deriváveis desta e do significado da igualdade =.

Exemplo: As seguintes inequações aparecem no trait ISEMPTY (sec. 3.2.22):

isempty(new())

not(isempty(insert(c,e)))

iii) *Cálculo de predicados de primeira ordem com Igualdade*. Th(T) contém os axiomas do cálculo de predicados de primeira ordem tipado com igualdade (propriedades reflexiva, simétrica, transitiva e sustituição) e, é fechada sob estas regras de inferência (ver [TUR 87]).

iv) *Indução*. Se um trait T tem a cláusula

S **generated by** [op1, op2, ..., opn]

e *P(s)* é uma fórmula bem-formada com a variável livre *s* de sort S, então Th(T) contém a fbf:

∀[s:S] **P(s)**

se para cada operador **opi** em [op1, op2, ..., opn] a fbf:

Qi ==> P(opi(x1, ...,xk))

está em Th(T), onde:

k é a aridade do operador opi,

xj são variáveis que não aparecem livres em P,

Qi é a conjunção de P(xj), para cada j tal que o j-ésimo argumento de **opi** é de sort S.

A cláusula **generated by** será considerada posteriormente na parte 2.1.2.2.

v) *Redução*. Se um trait T tem a cláusula

S partitioned by [op1, op2, ..., opn]

então Th(T) contém a fbf:

∀[s1,s2:S] (Q ==> s1=s2) , onde:

Q é a conjunção (com **opi** em [op1, ..., opn] e j tal que o j-ésimo argumento de **opi** é de sort S) de:

∀[x1:S1, ...,xk:Sk] (subst(opi,j,s1)=subst(opi,j,s2))

onde:

S1, ..., Sk é o domínio de **opi**, e

subst(op,j,s) é op(x1, ..., xk) com xj substituído por s.

Exemplo: O trait BinaryTree ([GUT 85], pp.80) contém a cláusula

C partitioned by [left, right, content, isLeaf] .

Se Q = ∀[c:C] subst(left,1,c1) = subst(left,1,c2) and
subst(right,1,c1) = subst(right,1,c2) and
subst(content,1,c1) = subst(content,1,c2) and
subst(isLeaf,1,c1) = subst(isLeaf,1,c2)

i.e., Q = ∀[c:C] left(c1) = left(c2) and
right(c1) = right(c2) and
content(c1) = content(c2) and
isLeaf(c1) = isLeaf(c2) ,

então Th(BinaryTree) contém a fórmula bem-formada:

∀[c1,c2:C] Q ==> c1 = c2

Definição. A parte final de um trait, a partir da cláusula **constrains** em diante (*propPart*), está *PROPRIAMENTE RESTRITA* se implica (descreve) propriedades de somente operadores aparecendo na lista da cláusula *constrains*.

Exemplo 1.

```
SINGLETON1 : trait
     assumes CONTAINER
     introduces
        singleton : E --------> C
     constrains singleton so that                    (*)
     for all [ e:E]
         singleton(e) = insert(new(),e)              (**)
```

Exemplo 2.

```
SINGLETON2 : trait
     assumes CONTAINER
     includes SIZE
     introduces
        singleton : E --------> C
     constrains singleton so that                    (a)
     for all [ e:E]
               singleton(e) = insert(new(),e)
               size(singleton(e)) = 1                (b)
```

CONTAINER é um trait que especifica uma estrutura de dados que contém elementos (Ex. conjuntos, filas, pilhas, etc.) ao qual estão associados dois operadores: new e insert. A cláusula "**assumes** ..." e "**includes** ..." serão estudadas posteriormente nas partes 2.1.3.5 (c e d).

No primeiro exemplo, a parte comprendida entre (*) e (**) está propriamente restrita, pois descreve propriedades somente do operador *singleton*. No segundo exemplo, a parte comprendida entre (a) e (b), descreve propriedades dos operadores *singleton* e *size*, e este último não faz parte da lista da cláusula **constrains**, portanto, esta parte do trait SINGLETON2 não está propriamente restrita.

A ocorrência de um identificador de sort (**sortId**) na cláusula *constrains ...* representa a lista de todos os operadores da parte introdutoria (**opPart**) do trait principal cujas signaturas (domínios e imagens) incluem este identificador. Por exemplo, no trait **BinaryTree** de [GUT 85], pp.80:

```
BinaryTree: trait
     imports Cardinal
     introduces
          <#> : E     --------> C
        <#,#> : C, C  --------> C
         left : C     --------> C
        right : C     --------> C
         size : C     --------> Card
       isLeaf : C     --------> Bool
      content : C     --------> E
     constrains C so that
     ....................
```

A cláusula

constrains C **so that**

indica que o sort C representa os operadores <#>, <#,#>, left, right, size, isLeaf e content declarados na parte "**introduces**" com domínio e/ou contradomínio sendo C e, que aparecem nos axiomas do trait.

Definição. Seja T um trait e P a propPart:

constrains sortedOp*, **so that** props

P é *propriamente restrita* no trait T+P se e somente se, cada fbf na teoria associada com T+P está contida na teoria associada com T ou contém um operador listado em *sortedOp**,.

c) Extensão Conservativa

Definição. Uma teoría Th2 é uma **EXTENSAO CONSERVATIVA** da teoría Th1, denotada por Th1 ≤ Th2, se o conjunto de fórmulas bem-formadas de Th2, contendo somente operadores definidos em Th1, é exatamente a teoría Th1; em outras palavras, se Th2 restrita aos operadores e equações de Th1, é exatamente Th1 ([TUR 87]).

Th1 ≤ Th2 <==> ∀ op ∈ Th1 : Th2 = Th1

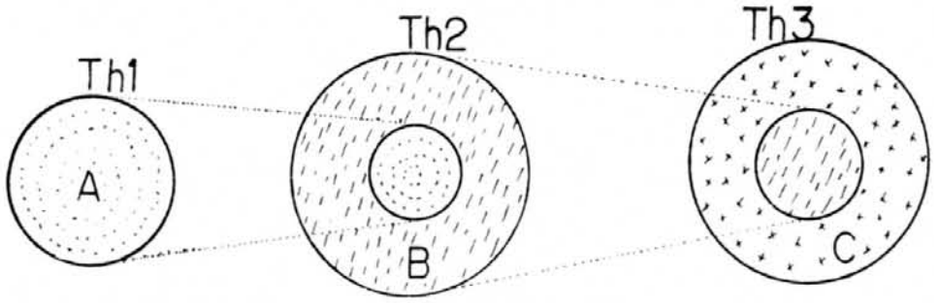

Fig. 2.4 Extensão Conservativa

Exemplo.

```
SIZE : trait
     imports  CARDINAL
     introduces
          new : {}   --------> C
       insert : C, E --------> C
         size : C    --------> Card
     constrains   size   so that
        C generated by [ new, insert ]
              size(new()) = 0
```

O conjunto de axiomas (equações) de SIZE está formado por

{ axiomas de CARDINAL } + { size(new()) = 0 }

logo a teoría de SIZE é uma extensão conservativa da teoría do trait importado CARDINAL .

2.1.3.3 Operadores Geradores (generated by)

Gramática:

generators ::= sortId **generated** bylist*

bylist ::= **by** [sortedOp*,]

sortedOp ::= signature

Verificação:

- A imagem de cada operador da lista *generated by* deve ser o sort *sortId* de generators (Fig. 2.6).
- Pelo menos um operador em cada lista de geradores deve ter um domínio onde sort gerado não ocorre.

sintaxe: A sintaxe dos operadores geradores é mostrada na Fig. 2.5

A presença da cláusula S2 **generated by [op1, ..., opn]** em um trait, significa que cada termo do sort S2 é igual a algum termo formado com os operadores op1, ..., opn, i.e., os operadores *opi*, i=1..n, são os <u>únicos</u> com imagem o sort S2.

```
┌──────────────────────────────────────────┐
│  introduces                              │
│      op1 : A --------> S2                │
│      op2 : A, S2 ----> S2                │
│      .....................               │
│      opn : B, X -----> S2                │
│  constrains . . . so that                │
│      S2 generated by [ op1, op2, ..., opn ] │
│  ........................................ │
└──────────────────────────────────────────┘
```

Fig. 2.5 Generated by

A cláusula <u>generated</u> <u>by</u> introduz uma regra de inferência indutiva que pode ser usada para provar

propriedades que são verdadeiras para todos os termos de sort S2.

generated by afirma que o conjunto { op1, op2,..., opn } contém suficientes operadores para gerar todos os

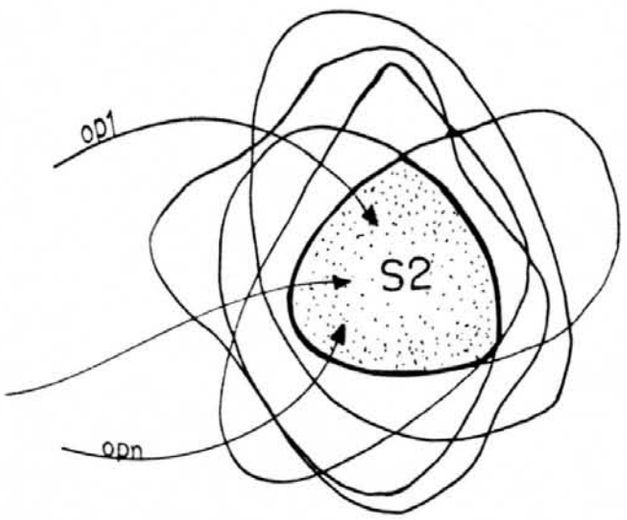

Fig. 2.6 Operadores Geradores

valores do sort S2, i.e., op1, op2, ..., opn são os únicos **construtores** de S2. Pelo menos um operador **opi** deve ter um domínio no qual S2 não faz parte. Em 2.1.4.2 será mostrado o uso dos operadores geradores e da cláusula "generated by".

Exemplo .

```
CONTAINER : trait
     introduces
         new : {}    --------> C
       insert : C, E --------> C
     constrains C so that
         C generated by [ new, insert ]
```

Neste exemplo, os únicos geradores do sort C são os operadores new e insert.

A cláusula *"sortId generated by [...]"* é uma maneira de especificar teorias grandes e de caracterizar a noção de "todos os sortId's".

A inclusão de *generated by* permite provar axiomas usando indução completa ([DAH 89]) e todas as provas estão baseadas na substituição equacional direta, i.e., nada pode ser deduzido a partir da ausência de equações ([LIS 86]).

2.1.3.4 Operadores Observadores (partitioned by)

Gramática:

```
partitions ::= sortId partitioned bylist*
    bylist ::= by [ sortedOp*, ]
  sortedOp ::= signature
```

Verificação:

- O domínio de cada operador da lista *partitioned by* deve incluír o sort *sortId* de *partitions (Fig. 2.8).*
- A imagem de pelo menos um operador em cada bylist deve ser diferente do sort particionado *sortId* de *partitions*.

sintaxe (Fig. 2.7) :

```
introduces
    op1 : S1 --------> A
    op2 : S1, E -----> B
    ....................
    opm : S1 --------> X
constrains . . .  so that
    S1 partitioned by [ op1, op2, ..., opm ]
.........................................
```

Fig. 2.7 A cláusula partitioned by

A presença da cláusula **S1 partitioned by [op1, ..., opn]** em um trait, significa que cada termo do sort S1 é igual a algum termo no qual op1, op2, ..., opm são os únicos operadores com domínio S1. A imagem de pelo menos um operador op1 deve ser diferente de S1.

A cláusula partitioned by indica que op1, op2, ... e opm são suficientes para distinguir os termos diferentes de

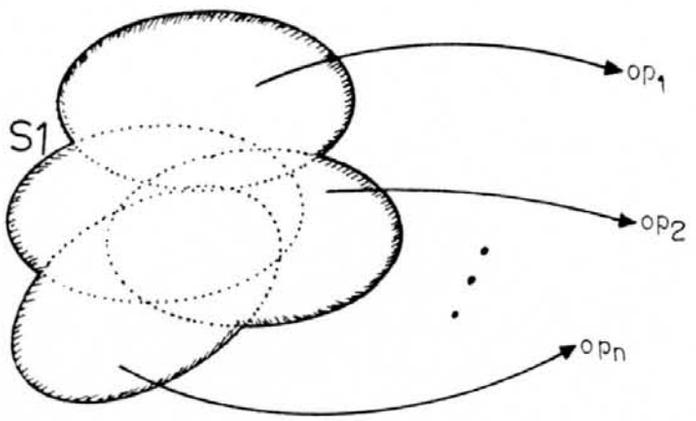

Fig. 2.8 Operadores Observadores

sort S1 : se dois termos de sort S1 não são iguais, a diferença pode ser observada usando um operador de { op1, op2, ..., opm } .

partitioned by indica que op1, op2, ... e opm formam um conjunto completo de operadores observadores para o sort S1. Isto quer dizer que, para qualquer termos t1 e t2, se as igualdades op1(t1)=op1(t2), op2(t1)=op2(t2), ..., opm(t1)=opm(t2) , são todas válidas, então pode-se concluir que t1 = t2.

Exemplo :

```
PAIR : trait
     introduces
          p : T1, T2 --------> C
      first : C      --------> T1
     second : C      --------> T2
     asserts
          C generated by [ p ]
          C partitioned by [ first, second ]
          for all [ f:T1, s:T2 ]
               first( p(f,s) ) = f
               second( p(f,s) ) = s
```

Sejam f1,f2∈T1 e s1,s2∈T2 tal que

p(f1,s1) = c1 e,

p(f2,s2) = c2

onde p, first e second são os operadores definidos no trait PAIR. Se first(c1) = first(c2) e second(c1) = second(c2); então a cláusula

```
          C partitioned by [ first, second ]
```

permite afirmar com certeza que c1 = c2.

2.1.3.5 Referências Externas

Gramática:

```
          trait ::= traitId : trait traitBody finals
      traitBody ::= externals simpleTrait
      externals ::= (assumes) (imports) (includes)
        assumes ::= assumes traitRef*,
        imports ::= imports traitRef*,
       includes ::= includes traitRef*,
       traitRef ::= traitId ( renaming )
```

Verificação :

- Nenhuma referência externa é recursiva; i.e., o nome

ou identificador de um trait T NAO pode aparecer numa das suas referências externas nem nas referências externas de outro trait referenciado externamente por T.

Exemplo: É incorreto a especificação dos seguintes traits:

```
T1: trait
     assumes T1, Tw        <-- Referência recursiva de T1
     imports T2, Tv        <-- T2 inclui a T1
     . . . . . .
T2 : trait
     includes  Tx, T1, Ty   <-- T1 inclui a T2
     . . . . . .
```

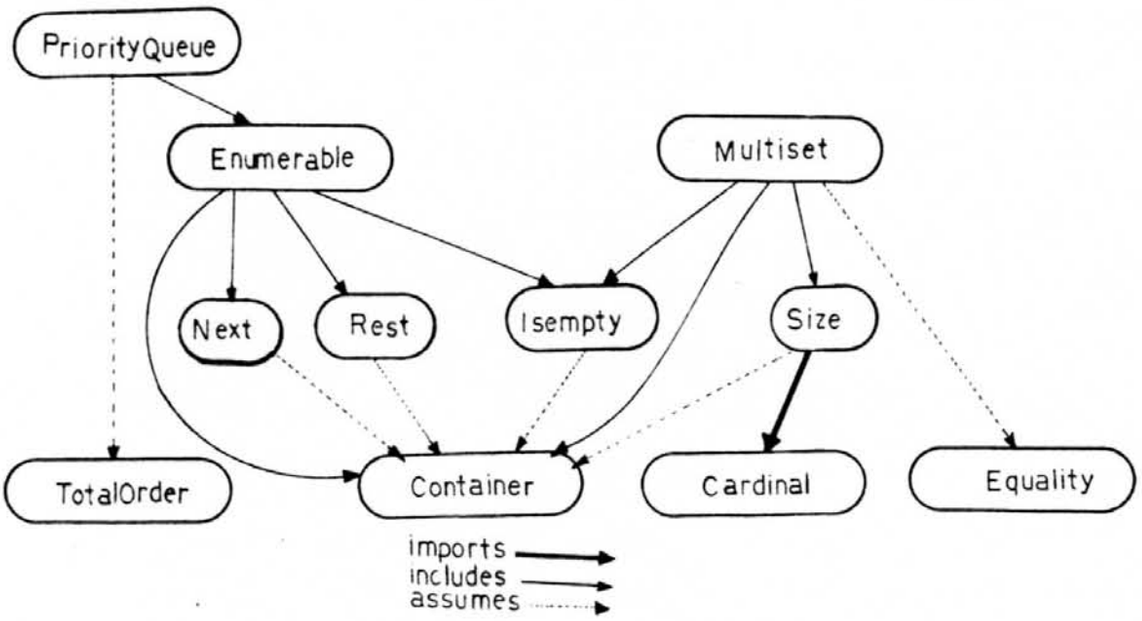

Fig. 2.9 Referências Externas: relações entre traits

Daqui por diante, **traitId** será o identificador do "trait principal" (importador) e cada **traitRef** será o identificador de um "trait secundário" (importado, incluido). Na Fig. 2.9, traitId é o identificador para Multiset, e traitRef é o identicador para Isempty, Size, Container e Equality.

a) Mecanismo de Renomeação

Gramática:

```
     traitRef ::= traitId (renaming)
     renaming ::= with [ ( opRename | sortRename )*, ]
     opRename ::= opId for oldOp
   sortRename ::= sortId for oldSort
        oldOp ::= sortedOp
      oldSort ::= sortId
```

Verificação:

- Nenhum operador *sortedOp* pode ocorrer mais de uma vez como um operador *oldOp*.
- Nenhum sort *sortId* pode ocorrer mais de uma vez como um sort *oldSort*.
- Cada sort a ser renomeado (*oldSort*) deve aparecer na signatura do trait rotulado por *traitId*.
- Deve existir um único mapeamento dos operadores a serem renomeados (*oldOp's*) para a signatura

  Sig = < (s_dom, s_range), (opId) >

do trait rotulado por *traitId*, tal que para cada par **<oldOp,Sig>**:

i.) Os *opId's* se igualam.

ii.) Se o operador a ser renomeado (*oldOp*) inclui um domínio, este é o mesmo que *s_dom* de Sig.

iii.) Se o operador a ser renomeado (*oldOp*) inclui um contradomínio, este é o mesmo que *s_range* de Sig.

Definição. A **trasladação** ou **tradução** de um trait T com referências externas (imports, includes, assumes), significa transformar o trait T , usando o mecanismo de renomeação, em outro trait equivalente sem referências externas.

O mecanismo de renomeação funciona da seguinte

maneira:

A trasladação do trait rotulado por **traitId** de traitRef é realizada aplicando primeiro a operadores (*opRenames*) e depois a sorts (*sortRenames*):

- Simultâneamente, para cada operador opRename, substituir a parte *opId* de cada ocorrência da signatura Sig na qual *oldOp* é mapeado, por *opId* de *opRename*.

- Logo então, simultâneamente para cada sort *sortRename*, substituir cada ocorrência do sort *oldSort* por seu respectivo identificador novo *sortId*.

Os traits nunca são parametrizados explícitamente, no entanto, o mecanismo de renomeação, permite que quaisquer entidades de um trait (sorts, operadores) sejam parâmetros potenciais.

Exemplo :

```
CONTAINER : trait
     introduces
          new : {}   --------> C
       insert : C, E --------> C
     constrains C so that
          C generated by [ new, insert ]
POINTWISEIMAGE : trait
     assumes CONTAINER with [ DC for C, DE for E ]
             CONTAINER with [ RC for C, RE for E ]
     introduces
          extOp : DC --------> RC
       pointOp : DE --------> RE
     constrains . . . .
```

A cláusula

CONTAINER **with** [DC **for** C, DE **for** E]

significa "trocar cada ocorrência de C no trait CONTAINER por DC e cada ocorrência de E em CONTAINER por DE".

Observe que, no trait POINTWISEIMAGE, o mecanismo de renomeação foi aplicado duas vezes e simultâneamente a sorts definidos no trait CONTAINER.

O mecanismo de renomeação transforma o trait *POINTWISEIMAGE* em uma das seguintes formas:

i. Se RC ⊆ DC:

POINTWISEIMAGE : **trait**
introduces
new : { } --------> DC
insert : DC, DE --------> DC
extOp : DC --------> RC
pointOp : DE --------> RE
constrains .....

ii. Se DC ⊆ RC:

POINTWISEIMAGE : **trait**
introduces
new : { } --------> RC
insert : RC, RE --------> RC
extOp : DC --------> RC
pointOp : DE --------> RE
constrains .....

Nos dois exemplos anteriores, os geradores de DC e RC são new e insert.

b) IMPORTS (Combinação de traits independentes)

A sintaxe do mecanismo de importação de traits é mostrada na Fig. 2.10

A cláusula **imports** é o mecanismo que permite a importação de um trait e sua teoría associada (Fig. 2.11). A importação é usada para:

- fazer que a estrutura de uma especificação seja mais fácil de ler.
- introduzir verificação adicional.

Os operadores que aparecem num trait importado não são restritos (modificados) pelo trait importador nem por outros traits importados. Isto assegura que:

A teoría associada a um trait T é uma extensão conservativa da teoría associada a cada teoría dos traits importados por T.

```
traitId : trait
    imports  traitId1 ( with [
                              sortId1 for oldSort1,
                              . . . . . .
                              sortIdn for oldSortn;
                              opId1 for oldOp1,
                              . . . . . .
                              opIdm for oldOpm ] )
             . . . . . .
             traitIdn (  . . . )
      introduces . . .
```

Fig. 2.10 Importação de Traits: sintaxe

Este principio permite que dois traits importados por um mesmo trait T, não interfiram um com outro, sob nenhuma situação inesperada. A compreensão dos operadores de cada trait importado independe do contexto no qual o trait é importado.

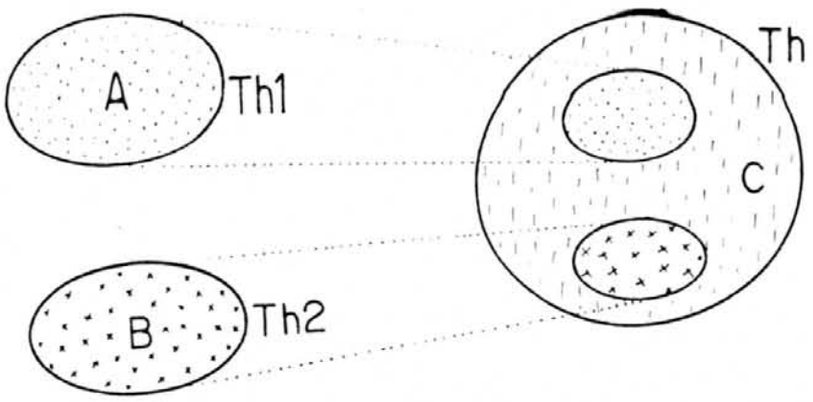

Fig. 2.11 Mecanismo de Importação em Larch

Exemplo 1.

```
SIZE : trait
     imports  CARDINAL
     introduces
          new : {}    --------> C
       insert : C, E  --------> C
         size : C     --------> Card
     constrains   size   so that
        C generated by [ new, insert ]
              size(new()) = 0
```

Pela cláusula "*imports CARDINAL*", o trait SIZE, está importando os operadores 1, +, * e - junto com os axiomas e restrições associados a estes operadores, mais o trait ORDINAL que é importado por CARDINAL.

Exemplo 2.

```
COLLECTIONS_EXTENSIONS : trait
   assumes ELEMENT_EQUALITY,
           CONTAINER with [ {} for new ]
   imports ISEMPTY with [ {} for new ]
           SINGLETON with [ {} for new, {#} for singleton ],
           CONTAINMENT with [{} for new ],
           JOIN with [ {} for new, U for .join ]
   . . . . . .
```

A primeira parte do trait COLLECTIONS_EXTENSIONS mostra a importação de traits usando o mecanismo de renomeação para **operadores**.

Exemplo 3.

```
BASICGRAPH : trait
  assumes EQUALITY with [Node for T ]
  imports SET with [ NodeSet for C, Node for E ]
          PAIR with [ Edge for C, Node for T1, Node for T2 ]
  introduces
      . . . . . .
```

Este trait mostra a importação de traits usando o mecanismo de renomeação para **sorts** (NodeSet for C, Node for E, etc.). Isto permite usar o sort C simultâneamente como NodeSet e Edge no trait BASICGRAPH.

c) INCLUDES (Combinação de traits interativas)

Sintaxe: ver Fig. 2.12.

A cláusula **includes** . . . permite combinar diversos traits com diferentes tratamentos de um mesmo operador.

```
traitId : trait
    includes  traitId1 { with [
                              sortId1 for oldSort1,
                              . . . . . .
                              sortIdn for oldSortn;
                              opId1 for oldOp1,
                              . . . . . .
                              opIdm for oldOpm ] }
              . . . . . .
              traitIdn {  . . . }
      introduces . . .
```

Fig. 2.12 Inclusão de Traits: sintaxe

Qualquer importação de um trait "correto" Tc pode ser substituído pelo mecanismo de inclusão (includes) sem alterar sua teoría associada ou torná-lo incorreto (o trait).

A diferença entre os mecanismos de IMPORTAÇAO e INCLUSAO é que, no primeiro, incorpora-se uma teoría não

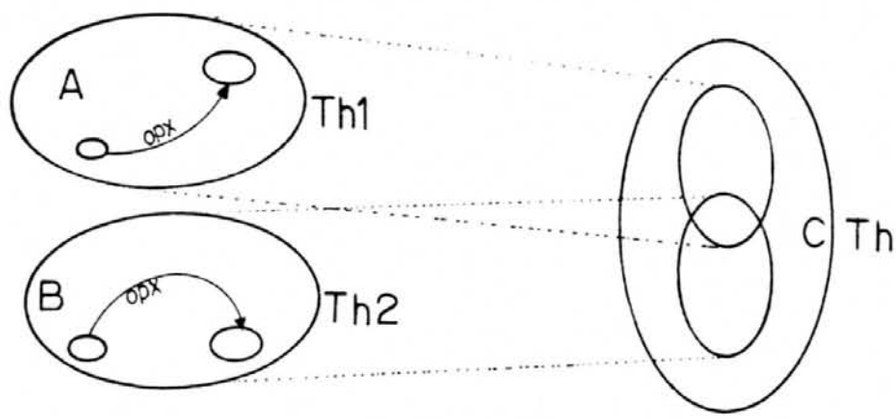

Fig. 2.13 Mecanismo de Inclusão

modificável (os operadores importados não podem ser restritos pelo trait importador), e no segundo, a teoría incluída pode ser modificada mesmo que a verificação não assegure a compreensão independente do contexto (trait importador) no qual é usado (Fig. 2.13).

Exemplo 1. Observe os quatro traits seguintes:

```
REFLEXIVE : trait          % relação reflexiva %
     introduces
          rel : T, T --------> Bool
     constrains  rel so that for all [ t:T ]
               rel(t,t) = true

SYMMETRIC : trait          % relação simétrica %
     introduces
          rel : T, T --------> Bool
     constrains  rel so that for all [ t1,t2:T ]
               rel(t1,t2) = rel(t2,t1)

TRANSITIVE : trait         % relação transitiva %
     introduces
          rel : T, T --------> Bool
     constrains  rel so that for all [ t1,t2,t3:T ]
           and(rel(t1,t2), rel(t2,t3)) ==> rel(t1,t3) = true


EQUIVALENCE : trait        % relação de equivalencia %
     includes  REFLEXIVE, SYMMETRIC, TRANSITIVE
```

Neste exemplo, o trait EQUIVALENCE, através do mecanismo de inclusão está incorporando a teoría (operadores, axiomas,etc.) dos traits REFLEXIVE, SYMMETRIC e TRANSITIVE. Observe que a teoría associada a REFLEXIVE (o único axioma

relacionado com o operador *rel*) é modificada implícitamente (ampliada) para três axiomas, i.é., o significado do operador *rel* é ampliado:

```
rel(t1,t2) = true
rel(t1,t2) = rel(t2,t1)
rel(t1,t2) & rel(t2,t3) ==> rel(t1,t3) = true
```

Exemplo 2.

```
ENUMERABLE : trait
     includes  CONTAINER, NEXT, REST, ISEMPTY
     constrains C so that
          C partitioned by [ next, rest, isempty ]

PRIORITYQUEUE : TRAIT
     assumes TOTALORDER with [ E for T ]
     includes ENUMERABLE
     constrains next, rest, insert so that
          for all [ q:C; e:E ]
          next(insert(q,e))= if isempty(q)
               then e
               else if next(q) ≤ e then next(q)
                                   else e
          rest(insert(q,e))= if isempty(q)
               then new
               else if next(q) ≤ e then insert(rest(q),e)
                                   else q
```

Os axiomas que modificam o operador isempty em PRIORITYQUEUE são herdados do trait ENUMERABLE, que por sua vez, herdou do trait ISEMPTY. Observe que, tanto *next* como *rest* (definidos em ENUMERABLE) são modificados pelos axiomas de PRIORITYQUEUE.

d) Suposições (assumes)

Definição. Seja A(T) o conjunto de todas as cláusulas *assumes ...* dos traits importados ou incluídos no trait T, e R(T) o resultado da tradução de T depois de remover os assumes. A(T) é **liberado** ou **descarregado** por T, se a teoría associada com a tradução de cada "traitId **with** [...]" de A(T) é um subconjunto da teoría associada com R(T).

Exemplo : Seja T o seguinte trait:

```
COERCECONTAINER1 : trait
     assumes CONTAINER with [ DC for C ]
             CONTAINER with [ RC for C ]
     introduces
          coerce : DC --------> RC
     constrains coerce so that for all [ dc :DC; e:E ]
          coerce(new()) = new()
          coerce(insert(dc,e)) = insert(coerce(dc),e)
```

```
A(T) = {   assumes CONTAINER with [ DC for C ]
           assumes CONTAINER with [ RC for C ]   }
```

R(T) é o seguinte trait:

```
COERCECONTAINER2 : trait
     introduces
             new : {}    --------> RC
          insert : DC, E --------> RC
          coerce : DC    --------> RC
     constrains coerce so that for all [ dc :DC; e:E ]
          coerce(new()) = new()
          coerce(insert(dc,e)) = insert(coerce(dc),e)
```

```
traitId : trait
    assumes  traitId1 ( with [
                              sortId1 for oldSort1,
                              . . . . . .
                              sortIdn for oldSortn;
                              opId1 for oldOp1,
                              . . . . . .
                              opIdm for oldOpm ] )
             . . . . . .
             traitIdn (  . . . )
      introduces . . .
```

Fig. 2.14 A cláusula Assumes: sintaxe

Quase sempre constroi-se muitas especificações gerais que posteriormente são especializadas em diferentes formas. Considere o seguinte exemplo modificado de [GUT 85],pp.32 (Fig. 2.15) :

```
BAG : trait
    introduces
          {} :                 --------> Bag
       insert : Bag, Element --------> Bag
       delete : Bag, Element --------> Bag
    element? : Bag, Element --------> Bool
    constrains {}, insert, delete, element? so that
       Bag generated by [ {}, insert ]
       Bag partitioned by [ delete ]
       for all [ b:Bag; e1,e2 : Element ]

          element?({},e1) = false
          element?(insert(b,e2),e1)  =  (e1=e2)  |
                                        (element?(b,e1))
          delete({},e1) = {}
```

```
delete(insert(b,e1),e2) = if e1=e2
                          then b
                          else insert(delete(b,e2),e1)
```

Pode-se especializar esta especificação para INTBAG,usando o mecanismo de renomeação **Integer** por **Element** e incluir isto no trait no qual operadores tratando com Integer são especificados, por exemplo:

```
INTBAG : trait
    imports INTEGER
    includes BAG with [ Integer for Element ]
```

As interações entre BAG e INTEGER são muito limitadas. Nada em BAG faz qualquer suposição sobre o significado dos operadores que ocorrem em INTEGER : 0, +, e <. Considere agora, a extensão de BAG para BAG1 agregando o operador rangeCount:

```
BAG1 : trait
  imports BAG, CARDINAL
  introduces
      rangeCount : Bag, Element, Element --------> Integer
              <  : Element, Element      --------> Bool
  constrains rangeCount so that for all [ e1,e2,e3:Element;
                                          b: Bag ]
     rangeCount({},e1,e2) = 0
     rangeCount(insert(b,e3),e1,e2) = rangeCount(b,e1,e2) +
                ( if (e1 < e3) & (e3 < e2) then 1 else 0 )
```

BAG1 não faz suposições sobre as propriedades do operador <. Pode-se ter idéias definidas sobre as propriedades que < deve ter em qualquer especialização, por exemplo, ordem total. Esta restrição pode ser especificada com a seguinte suposição:

```
BAG2 : trait
      assumes ORDERED with [ Element for T ]
      imports BAG, CARDINAL

      introduces
         rangeCount : Bag, Element, Element --------> Integer
      constrains rangeCount so that for all [ e1,e2,e3:Element;
                                              b: Bag ]
       rangeCount({},e1,e2) = 0
       rangeCount(insert(b,e3),e1,e2) = rangeCount(b,e1,e2) +
                  ( if (e1 < e3) & (e3 < e2) then 1 else 0 )
```

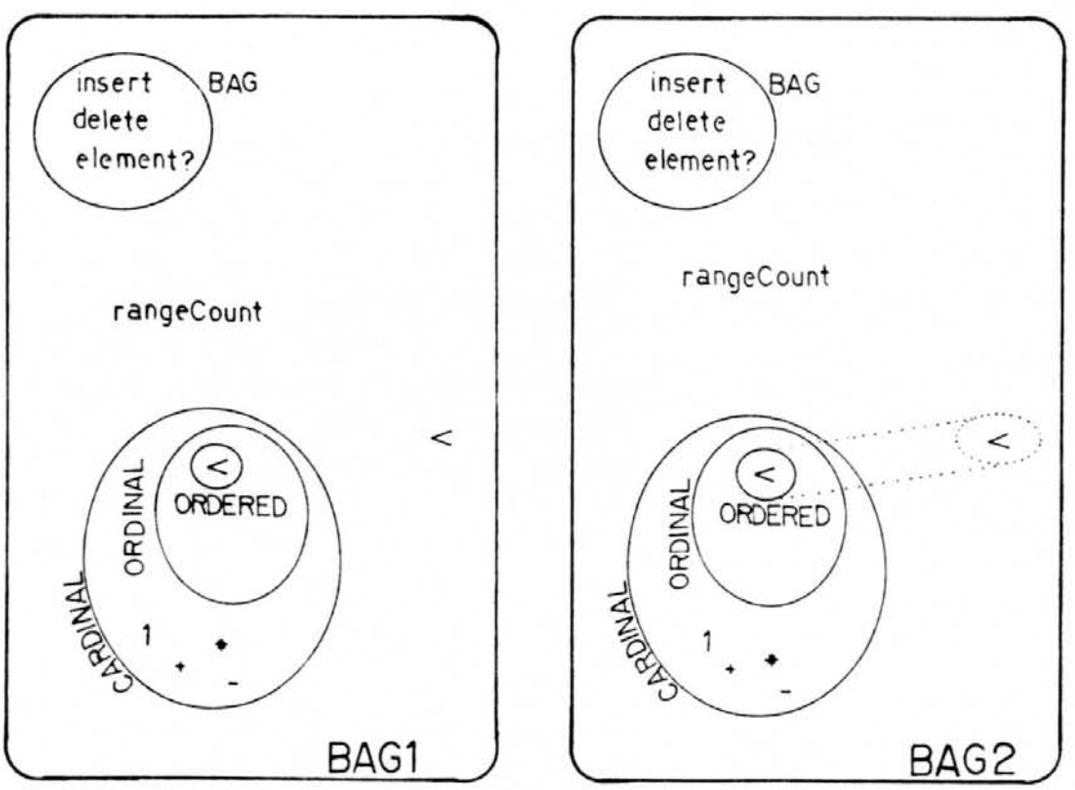

Fig. 2.15 A cláusula 'assumes' nos traits BAG1 e BAG2

A teoría associada com BAG2 é a mesma como se:

```
ORDERED with [ Element for T ]
```

fosse incluída. Isto podería ser usado para derivar várias

propriedades de BAG2, por exemplo, que rangeCount é monotonico no seus últimos argumentos.

Quando BAG2 é importado ou incluído em outro trait, a suposição tem que ser necessariamente descarregada. Em: *BAG1* : **trait**

assumes BAG2 **with** [Element **for** T]

imports INTEGER

o trait (BAG1) significará mostrar que a teoría associada (renomeada) com ORDERED é um subconjunto da teoría associada com INTEGER.

Com frequência, as suposições (assumes) de um trait são usadas para descarregar as suposições dos traits importados ou incluídos.

2.1.3.6 Implicações (consequências), Conversoes e Exceções

Gramática:

```
        trait ::= traitId : trait traitBody  finals
       finals ::= ( consequences ) ( exempts)
 consequences ::= implies conseqProps ( converts )
  conseqProps ::= traitRef*,  props
        propos ::= generators*  partitions*  axioms*
     converts ::= converts conversion*,
   conversion ::= [ sortedOp*, ]
      exempts ::= exempts exemptTerms*
  exemptTerms ::= ( for all [ varDcl*, ] ) term*
```

Verificação:

- Se props é agregado a propPart (a semântica do trait principal), o trait resultante deve satisfazer a

verificação de 2.1.3.1, 2.1.3.2, 2.1.3.3 e 2.1.3.4
- Cada termo deve satisfazer a verificação de 2.1.3.1, 2.1.3.2, 2.1.3.3 e 2.1.3.4 .

Nesta parte da linguagem, introduzimos REDUNDANCIA VERIFICAVEL : suposições são verificadas quando um trait é incluído ou importado, e conjuntos de restrições (**constrains**) são verificadas contra os axiomas associados a cada cláusula **constrains**. A redundância verificável é introduzida na forma de asserções (afirmações) ao redor da teoría associada a um trait.

Definição. Um trait T *implica* suas consequências se a teoría associada com o conjunto de traits referenciados após a palavra chave **implies** (conseqProps) é um subconjunto da teoría associada com o trait T e o conjunto de operadores [**sortedOp***,] em cada **converts** é conversível.

Conversibilidade é definida usando a teoría e os *exempts* (conjunto de termos não restritos) do trait.

A teoría associada com a cláusula

implies traitRef*,
converts [sortedOp*,]

é a teoría associada com a parte

includes traitRef*,
introduces
{ Declaração de Operadores }
(**asserts** | **constrains**)
generators* partitions* axioms*

do trait onde as implicações (**implies** ...) aparecem.

Seja C uma conversão (*[sortedOp*,]*). Para cada termo t, que não contém variáveis de qualquer sort

aparecendo em uma cláusula ... ***generated by*** *byList**, em um trait principal T, a teoría de T deve :

- conter uma equação t=t1, onde t1 não contém operadores aparecendo em *sortedOp**, de C, ou
- conter uma equação t'=t1, onde t' é um subtermo de t, e t1 é uma instanciação de um termo aparecendo em ***exempts*** do trait T.

Existem duas classes de consequências (asserções):

a) Quando a teoría associada a um trait contém outra teoría. Isto é feito usando a cláusula **implies** . Considere, por exemplo, que ao trait BAG2, apresentado em 2.1.1.5.D, fosse agregado o seguinte:

```
implies for all [b:Bag; e1,e2,e3:Element ]
   ( e2 < e3 ) ==> ( rangeCount(b,e1,e2) ≤
                     rangeCount(b,e1,e3) )
```

implies pode ser usado para indicar consequências entendidas de uma especificação, tanto para a verificação como para aumentar o discernimento do leitor. A teoría a ser implicada pode ser especificada usando toda a potência da linguagem: utilizando **generated by** e **partitioned by** ou por referência a traits definidos em qualquer lugar.

A cláusula **implies** pode ter uma das seguintes formas:

```
implies traitId with [ ... for ..., ... ]
implies sortId generated by [ sortedOp, ... ]
implies sortId partitioned by [ sortedOp, ... ]
implies for all [ ..... ]   equações
implies converts . . .
```

b) Quando a teoría associada com um trait adequadamente define um conjunto de operadores em termos de outros operadores. Isto é feito utilizando **converts** [*sortedOp**,].

Converts é usado para dizer que a especificação define adequadamente uma coleção de operadores, i.e., que cada termo que não contém variáveis de qualquer sort aparecendo em uma cláusula **generated by** é provadamente igual a um termo que não contém qualquer dos operadores em *sortedOp**,.

Um problema comum com sistemas axiomáticos é decidir quando existem suficientes axiomas. **converts** providencia uma maneira de fazer uma sentença verificável em função de um conjunto de axiomas adequado. Considere o seguinte exemplo:

```
TABLESPEC : trait
    introduces
        new : {} ---> Table
        add : Table, Index, Val ---> Bool
         c  : Index, Table ---> Bool
       eval : Table, Index ---> Val
    isempty : Table ---> Bool
       size : Table ---> Card
    constrains new, add, c, eval, isempty, size so that
        for all [ ind, ind1: Index; val:Val; t:Table ]
              eval(add(t,ind,val),ind1)= if(ind=ind1)
                       then val
                       else eval(t,ind1)
              c(ind,new()) = false
              c(ind,  add(t,  ind1, val)) =  (ind=ind1)  or
                                             c(ind,t)
              size(new()) = 0
```

```
size(add(t, ind,val)) = if c(ind, t)
                        then size(t)
                        else size(t) + 1
isempty(t) = ( size(t) = 0 )
```

Considere a seguinte agregação a TABLESPEC :

converts [isempty] (*)

Isto diz, que termos tais como *isempty(new())* ou *isempty(add(new(), ind, val))* são provadamente iguais a termos que não contém *isempty*, por exemplo, isempty(t) é igual ao termo (size(t) = 0) que não contém o operador isempty.

Agora, substituir em TABLESPEC a agregação (*) pela seguinte:

converts [isempty, eval]

Termos contendo subtermos da forma *eval(new(),ind)* não são conversíveis a termos que não contém **eval**, de modo que uma mensagem de erro da forma:

ERROR: eval(new(),ind) not convertible ...

podería ser gerado. Esta incompleteza podería ser resolvida agregando outro axioma:

eval(new(),ind) = errorVal()

De qualquer maneira, isto requer o registro da decisão que podería ser não apropriada em tal trait desde que isto supusesse a existência de um operador errorVal para o sort Val. O problema fica resolvido com a cláusula **exempts** para indicar que a inconvertibilidade de alguns termos é aceita.

Se TABLESPEC for modificado para incluir:

exempts for all [ind: Index] eval(new(),ind)

a verificação associada com a cláusula **converts** requer agora que, para qualquer termo t que não contém variáveis de sort Table, a teoría associada a TABLESPEC deve conter :

* uma equação, t=t1, onde t1 não tem ocorrências de isempty ou eval, ou
* uma equação, t'=t1, onde t' é um subtermo de t e t1 é uma instanciação de eval(new(),ind).

Esta verificação assegura que cada termo contendo operadores na lista **converts** ou

- é definido pelos axiomas (em termos de operadores ausentes na lista), ou
- é explicitamente excluido.

2.1.3.7 Traits Incorporados Implicitamente

Três traits são incorporados implícitamente em outros traits: BOOLEAN, IFTHENELSE e EQUALITY. Isto é feito para assegurar a uniformidade em todos os operadores que podem ser restritos (modificados) pelos operadores destes traits.

```
BOOLEAN: trait
    introduces
        true: {} --------> Bool
       false : {} --------> Bool
         not : Bool --------> Bool
         and : Bool, Bool --------> Bool      %  &   %
          or : Bool, Bool --------> Bool      %  |   %
        impl : Bool, Bool --------> Bool      %  ==> %
        equi : Bool, Bool --------> Bool
    asserts Bool generated by [ true, false ]
      forl all [ b : Bool ]
        not(true()) = false()          % false        %
        not(false()) = true()          % true         %
```

```
        and(true(),b) = b                % true & b    %
        and(false(), b) = false()        % false & b   %
        or(true(),b) = true()            % true | b    %
        or(false(), b) = b               % false | b   %
        imp(true(),b) = true()           % true ==> b  %
        imp(false(), b) = b              % false ==> b %
        equi(true(),b) = true()          % true ≡ b    %
        equi(false(), b) = b             % false ≡ b   %
    implies converts [ not, and, or, imp, equi ]


IFTHENELSE : trait
    introduces
        ifthenelse : Bool, T, T --------> T
    asserts for all [ t1, t2 : T ]
        ifthenelse(true(),t1,t2) = t1
        ifthenelse(false(),t1,t2) = t2
    implies converts [ ifthenelse ]

EQUALITY: trait
    introduces
        equal :  T, T  --------> Bool     %   =    %
    asserts T partitioned by [ equal ]
        for all [ x, y, z : T ]
            equal(x,x)
            equal( equal(x,y), equal(y,x) )
            ( equal(x,y) & equal(y,z) ) ==> equal(x,z)
```

2.1.4 Consistência Interna e Completeza Suficiente da Linguagem Larch Compartilhada

Existem duas noções para avaliar uma especificação (trait):

- consistência interna, e
- completeza suficiente.

2.1.4.1 Consistência Interna

Definição. Um trait T é INCONSISTENTE INTERNAMENTE se a teoria Th(T) inclui a equação true = false. Se isto acontecer, então qualquer fórmula bem formada faz parte da teoria Th(T) e, a partir disto, qualquer axioma pode ser provado.

A inconsistência interna de um trait pode ser eliminada pela remoção dos axiomas que produzem a equação True() = False(), isto é, formando um conjunto apropriado (suficiente) de axiomas.

Exemplo: A seguinte especificação é inconsistente internamente:

```
1. INTBAG1 : trait
2.      imports INT
3.      introduces
4.            new : {} --------> IntBag
5.         insert : IntBag, Int --------> IntBag
6.         delete : IntBag, Int --------> IntBag
7.          is-in : IntBag, Int --------> Bool
8.      constrains new, insert, delete is-in so that
         for all [ b: IntBag; i,j: Int ]
9.         is-in(new(),i) = False()
10.        is-in(insert(b,i),j) = (i=j) or is-in(b,j)
11.        is-in(delete(b,i),j) = not(i=j) & is-in(b,j)
12.        delete(new(),i) = new()
```

13. delete(insert(b,i),j) = if (i=j)
then b
else insert(delete(b,j),i)

Seja o termo:

t = is-in(delete(insert(insert(new(),8),8),8),8)
= is-in(delete(b ,8),8)

Pelo axioma 11:

t = not(8=8) & is-in(b,8)
= False() & is-in(b,8)
= False() . . . . . (*)

Pelo axioma 13:

t = is-in(delete(insert(c,8),8),8) {c=insert(new(),8)}
= is-in(c,8)
= is-in(insert(new(),8) ,8)
= (2=2) or is-in(new(),8)
= True() or is-in(new(),8)
= True() . . . . . (**)

Pela transitividade da igualdade para (*) e (**) temos que a equação t = False() = True() faz parte de Th(INTBAG1), portanto o trait INTBAG1 é inconsistente internamente. É suficiente remover o axioma 11 para conseguir a consistência.

O problema é como saber, em tempo da especificação se temos um conjunto suficiente de axiomas de modo a obter-se uma especificação consistente internamente. A seleção de axiomas apropriados, depende muito do tipo que esta se especificando. Em [GUT 78] afirma-se que, é impossível dar um procedimento geral para construir axiomas para tipos algébricos.

Definição ([LIS 86] pp. 201). Um conjunto inicial de axiomas é dito **COMPLETO** se é impossível agregar um axioma independente, i.e., se todas as fórmulas bem formadas, ou

- são construídas a partir do conjunto inicial, ou
- são inconsistentes com o conjunto inicial.

No exemplo anterior, os axiomas 9, 10, 12 e 13 formam um conjunto completo de axiomas para INTBAG1, pois a inclusão de outro axioma (para os mesmos operadores) torna o trait inconsistente (por exemplo, o axioma 11).

2.1.4.2 Completeza Suficiente

No inicio da seção 2.1.2, mencionaba-se que uma forma de avaliar um trait era usando o conceito de Completeza Suficiente, i.e., o fato de saber quando o conjunto de axiomas de um trait é um numero suficiente de axiomas consistentes internamente. Ainda que J.Guttag a partir de 1978 introduzia a base teórica deste conceito ([GUT 78]), é a partir do Projeto Larch que foi implementado. A seguir apresenta-se algumas definições relacionadas ao conceito acima mencionado, e logo, define-se e mostra-se sua implementação em Larch Compartilhada.

DEFINICOES.

- Uma especificação é COMPLETA se todos seus modelos são isomórficos ([WIR 83]).

- Uma especificação S é COMPLETA, se e somente se, cada equação eq de S é deduzida de um modelo de S, usando as regras de reflexividade, simetria, transitividade, substituição, abstração e concretização ([CAS 88],[GOG 81]).

- COMPLETEZA de uma especificação algébrica é definida com respeito a algum domínio básico que se supõe descritos pelos os axiomas. Dado um modelo de um Domínio Básico, pode-se definir completeza em função de que, se algo é verdadeiro

no modelo, isto pode ser provado a partir dos axiomas ([LIS 86]).

Nenhuma das definições anteriores (orientadas a modelos) é apropriada para tratamento de especificações (traits) em Larch. A noção de COMPLETEZA SUFICIENTE é usada para traits que contém a cláusula *generated by*.

Uma cláusula *generated by* define um conjunto de funções que são suficientes para gerar todos os valores de um sort (ver Operadores Geradores).

Definição. Uma especificação é **SUFICIENTEMENTE COMPLETA** com respeito a uma cláusula *generated by* da forma

S **generated by** [f1,. . ., fn]

se é possível mostrar que

a. Todos os termos de variáveis livres de sort S que contém operadores de contradomínio S (fi: Dom ----> S), são provadamente iguais a termos onde as únicas funções com contradomínio S, são f1, ..., fn.
b. Todos os termos de variável livre que não são de sort S, são provadamente iguais a termos que não contém funções de contradomínio S.

A validez das duas condições acima implica em um conjunto de axiomas suficiente.

Considere a seguinte especificação (o exemplo anterior modificado) :

```
1. INTBAG2 : trait
2.      imports INT
3.      introduces
4.            new : {} --------> IntBag
5.         insert : IntBag, Int --------> IntBag
```

```
6.          delete : IntBag, Int --------> IntBag
7.           is-in : IntBag, Int --------> Bool

8.      constrains new, insert, delete is-in so that
9.      IntBag generated by [ new, insert ]
         for all [ b: IntBag; i,j: Int ]
10.        is-in(new(),i) = False()
11.        is-in(insert(b,i),j) = (i=j) or is-in(b,j)
12.        delete(new(),i) = new()
13.        delete(insert(b,i),j) = if (i=j)
                                   then b
                                   else insert(delete(b,j),i)
```

Para mostrar que esta especificação é suficientemente completa, precisa-se mostrar, por exemplo, que :

a. Qualquer termo da forma *delete(b,i)*, onde *b* e *i* são termos de variável livre, é igual a um termo onde *delete* não ocorre.

b. Qualquer termo da forma *is-in(b,i)*, onde *b* e *i* são termos de variável livre, é igual a um termo onde nem *new*, nem *insert* e nem *delete* ocorrem.

A prova de que *delete* pode ser removida (igualando a outro termo) é feita usando indução:

a) Passo Base:

Seja b = *new()*. Então, pelo axioma 12,

delete(new(), i) = new()

b) Passo de Indução:

Suponha que se a profundidade de aninhamento em b é menor que *n*, então *delete* pode ser eliminado (hipótese de indução).

Seja *insert(b1,i)* com profundidade de aninhamento n. Pelo axioma 13,

delete(insert(b1,i),j) = b1 OR insert(delete(b1,j),i)

A profundidade de aninhamento em ambos casos é menor que *n*, logo pela h.i., *delete* pode ser eliminado, e a prova fica terminada.

Agora provaremos que *is-in(b,i)* pode ser igual a outro termo onde *new*, *insert* e *delete* não ocorrem.

a) Passo Base:

Seja b = *new()*. Pelo axioma 10,

is-in(b,i) = is-in(new(),i) = False()

b) Passo de Indução:

Suponha que se a profundidade de aninhamento em b é menor que *n*, então *is-in* pode ser removido.

Seja *insert(b2,i)* com profundidade de aninhamento n.

Pelo axioma 11,

is-in(insert(b2,i),j) = (i=j) OR is-in(b2,j)

= (True() OR False()) OR is-in(b2,j)

Na primeira parte, is-in já foi removida. Na segunda parte, a profundidade de aninhamento de b2 é menor que n, logo pela h.i., is-in também pode ser removida.

2.2. LINGUAGEM DE INTERFACE LARCH (LIL)

2.2.1 Um Exemplo

Como introdução para esta segunda parte, apresenta-se a especificação do tipo **Stack** em Larch Compartilhada e Larch/Pascal e uma implementação em Linguagem Pascal.

a) Larch Compartilhada

```
LAST : trait
     imports NEXT with [ last for next]
ENUMERABLE : trait
     imports   ISEMPTY, LAST, REST
     includes CONTAINER
     constrains C so that
          C partitioned by [ rest, last, isempty ]
SSTACK : trait
     includes  ENUMERABLE with [ push for insert, top for
                          last, pop for rest, SStack for C,
                          StackItem for E ]
     constrains push, pop, top so that
          for all [ stk : SStack, e: StackItem ]
         top(push(stk,e)) = e
          pop(push(stk,e)) = stk
```

b) Larch/Pascal

```
TYPE Stack
     exports CreateS, IsemptyS, PushS, PopS, TopS
     based on sort SStack
         from SSTACK

FUNCTION CreateS( s: Stack) : Stack
     modifies nothing
     ensures  s_post = new()
FUNCTION IsemptyS(var s : Stack) : Boolean
     modifies nothing
     ensures if isempty(s) then True
                           else False
PROCEDURE PushS( s : Stack, e: StackItem)
     modifies at most [ s, top(s) ]
     ensures  s_post = push(s,e)
FUNCTION PopS(var s : Stack) : Stack
     requires isempty(s) = False
     modifies at most [ s, top(s) ]
     ensures  s_post = pop(s)
FUNCTION TopS(var s: Stack) : StackItem
     requires isempty(s) = False
     modifies nothing
     ensures StackItem = top(s)
```

c) Implementação

```
const
   maxstack = 100;

type
   stackitem = INTEGER;    (*  real, char, etc. *)
```

```
   stack = record
           item : array[1..maxstack] of stackitem;
           top  : 0 .. maxstack;
           end;
   s: stack;

function Create( s: stack) : stack;
   begin
     for i:= 0 to maxstack do s.item := 0;
     s.top := 0;
   end;
function EMPTY(s:stack) : boolean;
   begin
      if s.top = 0 then empty:= true
                   else empty:= false;
   end;  (* empty *)
procedure PUSH(var s:stack; x:stackitem);
  begin
        s.top := s.top + 1;
        s.item[s.top] := x
  end; (* push *)
function POP(var s:stack) : stackitem;
  begin
  if empty(s) then error('ERROR: stack is empty')
              else begin
                   pop:=s.item[s.top];
                   s.top:= s.top - 1
                   end;
  end; (* pop *)
function TOP(s:stack) : stackitem;
  begin
  if empty(s) then error('ERROR: stack is empty')
               else top := s.item[s.top];
  end;  (* top *)
```

2.2.2 Descrição das linguagens de Interface Larch

No momento da escrita deste texto, somente três linguagens de interface Larch foram definidas: Larch/CLU ([WIN 83]), Larch/PASCAL ([LIS 86]) e Larch/Ada-88 ([RAM 89]).

Cada Linguagem de Interface Larch é orientada a uma linguagem de programação específica, e esta tem influência desde os mecanismos de modularização até a escolha das palavras chaves ou reservadas na sua linguagem de Interface Larch associada.

O significado das palavras reservadas de uma linguagem de interface Larch, são derivadas diretamente do significado na linguagem de programação. Por exemplo, **var** em Larch/Pascal é derivada do significado de **var** na lista de parâmetros em Pascal; o significado de **signals** em Larch/CLU é derivado do significado de **signals** em CLU. Assim, a semântica de uma linguagem de Interface Larch é definida relativa a semântica da linguagem de programação. Dois objetivos são alcançados com isto:

- Entender com muita precisão o que significa uma implementação satisfazer uma especificação, e
- Garantir a correta tradução de uma linguagem de Interface Larch em cálculo de predicados.

Desde que, abstrações de dados (Tipos) incluem abstrações procedurais ([LIS 86]), neste texto, a descrição da linguagem de interface será feita em função do primeiro.

A especificação de um Tipo de Dado em qualquer

Linguagem de Interface Larch (LIL) tem três partes:

a) Um **cabeçalho** contendo o nome do tipo e os nomes das rotinas visíveis externamente.

```
Type typeId
        exports rotina1, ..., rotinaN
```

Exemplo: **Type** Stack
exports IsemptyS, PushS, PopS, TopS

b) Um **trait associado** junto com um mapeamento entre os tipos da especificação em LIL e os sorts no trait associado.

```
based on sort sortId
        from traitId with [ s for S, ... ]
```

sortId é o identificador do tipo que está se especificando. *traitId* é o nome do trait associado e fornece todas as signaturas (identificadores de sorts e operadores) que aparecem nas asserções da especificação de rotinas.

Exemplo:

```
based on sort SStack
        from SSTACK with [integer for StackItem]

mapeamento: SORT (LLC) <======> TIPO (LI)
            SStack     --------> Stack
            StackItem  --------> integer
```

A cláusula *based on sort* associa o tipo Stack com o sort SStack que aparece no trait STACK. Nesta especificação (Larch/Pascal), esta associação quer dizer que os termos da LLC de sort SStack serão usados para representar valores Pascal de tipo Stack. Por exemplo, o termo top(s) é usado para representar o valor que StackItem deve ter quando TopS retorna.

c) A **especificação interface de cada rotina** (módulo, procedimento, função, etc.) **do tipo**.

A especificação de uma rotina (de um tipo qualquer) é constituída de três partes:

a) O **cabeçalho** contendo o nome da rotina e os nomes e tipos dos parâmetros e valores retornados.

Exemplo:

Larch/Pascal:

```
Procedure PushS(var s:Stack; e:Integer)
Function PopS(var s:Stack) : Stack
```

Larch/CLU:

```
PushS = proc (s:Stack, e:Integer) returns (s:Stack)
PopS  = proc (s:Stack) returns (s1:Stack)
```

Numa especificação interface, dá-se significado aos nomes aparecendo em programas para relacioná-los aos nomes aparecendo nos traits. Assim, em uma especificação interface, são os nomes que ligam os traits na Linguagem Larch Compartilhada e os programas em sua respectiva linguagem de programação. Operadores (ex. push), sorts (ex. SStackm) e nomes de traits (ex. SStack) fornecem a ligação para a teoría definida pela coleção de traits. Nomes de rotinas (ex. PushS), parâmetros formais (ex. e) e tipos (ex. StackItem) providenciam a ligação para programas que implementam a especificação. É importante não confundir OPERADORES e SORTS (da Linguagem Larch Compartilhada) e ROTINAS e TIPOS (da linguagem de programação). Operadores e sorts aparecem em especificações, e não em programas. Rotinas e tipos aparecem em programas e não em traits (Fig.16).

Fig. 16 Relação LLC - LIL

b) Um **trait associado** que fornece a **teoria** dos operadores que aparecem no corpo da rotina.

Exemplo:

- `isempty(s) = false`
- S_{post} `= top(s)`
- **if** `isempty(s)` **then** `true`
  else `false`

Estes três axiomas aparecem no trait associado SSTACK.

c) O **corpo** da rotina que estabelece os requerimentos dos parâmetros da rotina e especifica os efeitos que a rotina deve produzir quando estes requerimentos são conseguidos.

requires *pre-condição*
modifies at most [*conj_var_modificáveis*]
ensures *post-condição*

O corpo da especificação de uma rotina coloca restrições sobre os argumentos com os quais a rotina pode propriamente ser chamada, e define os aspectos relevantes da organização da rotina quando esta é

propriamente chamada. Isto pode ser traduzido no seguinte estilo:

Requires Predicado ==> (**Modifies** Predicado
& **Ensures** Predicado)

A cláusula **requires** no corpo de uma rotina estabelece a *PRECONDIÇAO* que deve ser satisfeita em cada chamada. Um **requires** omitido é interpretado como verdadeiro.

O predicado **modifies at most** [v1, v2, ..., vn] afirma que a rotina não muda os valores de nenhuma variável da chamada, exceto, possivelmente algum SUBCONJUNTO das variáveis denotadas pelos elementos do conjunto {v1, v2, ..., vn }. Observar que este predicado é realmente uma asserção acerca das variáveis que NAO aparecem na lista, antes que, sobre o que acontece com uma variável que sim esta na lista. Portanto, somente as variaveis aparecendo na cláusula **modifies at most** podem ser modificadas. Uma variação desta cláusula é **modifies nothing**, para indicar que nenhuma variavel deverá ser modificada nesta rotina.

modifies at most [...] é um predicado embutido de uma linguagem de programação especifica. Cada linguagem de interface Larch é equipada com seu próprio conjunto de predicados incorporados (built-in).

A distinção entre valor inicial e final de uma variável é feita usando um identificador de variável subscrito:

pre para o valor inicial, e
post para o valor final.

A afirmação $v_{pre} = v_{post}$ significa que o valor da variável v não é alterado.

Devido a especificação interface não especificar a REPRESENTAÇAO do tipo ou, os ALGORITMOS nas rotinas, ainda outra ligação com o projeto é necessária. Devido a que esta ligação é oculta dos usuários do tipo de dado, o projeto pode ser modificado sem afetar sua correteza.

A especificação de cada rotina em uma linguagem de interface pode ser compreendida sem referenciar as especificações de outras rotinas. Isto em contraste com os traits, onde a especificação restringe os operadores por meio de relações entre traits (imports, includes, assumes).

2.2.3 Const rução Incremental de uma Especificação Interface

2.2.3.1 Estrategia ([WIN 87]):

Projeto Top-Down.

a) Desenvolver uma idéia aproximada do problema a ser desenvolvido baseado numa interação bem próxima com o proprietário do problema.
b) Decidir sobre as maiores abstrações:
   i. Linguagem de interface: Escrever os cabeçalhos dos componentes da linguagem de interface.
   ii. Linguagem Larch Compartilhada: Escrever a informação sintática dos traits da especificação: identificadores de sort, identificadores de operadores e signaturas.
c) Preencher os espaços em branco:
   i. Completar o corpo dos componentes da linguagem de interface, escrevendo asserções nos corpos das especificações das rotinas.
   ii. Definir a ligação explícita entre componentes LIL e LLC.
d) Verificar a compreensão do problema e sua

formalização. Repetir os passos a), b) e c) até completar a especificação.

2.2.3.2 Exemplo.

A seguir apresenta-se uma série de instantâneos que mostram a construção incremental de uma especificação em uma linguagem de Interface Larch (PASCAL). Supunha-se que se quer especificar o tipo Stack, e supõe-se o uso de um Editor de Especificação dirigido pela Sintaxe (EEDS), que mostre na tela os templates das figuras abaixo, e o cursor em cada ..... Começamos com a escrita dos nomes nos cabeçalhos do tipo a especificar (Stack e SSTACK).

```
┌── Type Stack                ‖   SSTACK : trait ──────┐
|       exports .....         ‖      introduces         |
|       based on sort .....   ‖           .....         |
|             from  SSTACK.   ‖      constrains         |
|                             ‖           .....         |
└── Fig. 2.17 Exemplo de Especif. em Larch/Pascal: Etapa 1 ┘
```

A fig. 2.18, mostra o preenchimento dos cabeçalhos de cada rotina escrita na cláusula **exports** . Cada uma com seus respetivos parâmetros de entrada e saída.

```
┌──────────────────────────────────────────────────────┐
|Type Stack                   ‖                        |
|     exports CreateS, IsemptyS, ‖   SSTACK : trait    |
|              PushS,PopS,Tops   ‖     introduces       |
|     based on sort .....        ‖      .....           |
|           from  SSTACK.        ‖     constrains       |
|                                ‖      .....           |
```

```
FUNCTION CreateS( s: Stack) : Stack
     requires .....
     modifies at most .....
     ensures .....
```

```
FUNCTION IsemptyS(var s : Stack) : Boolean
     requires .........
     modifies at most ......
     ensures ..........
PROCEDURE PushS( s : Stack, e: StackItem)
     requires .........
     modifies at most ......
     ensures ..........
FUNCTION PopS(var s : Stack) : Stack
     requires .......
     modifies at most  .........
     ensures  .......
FUNCTION TopS(var s: Stack) : StackItem
     requires .........
     modifies at most ........
     ensures ........
```

Fig. 2.18 Exemplo de Especif. em Larch/Pascal: Etapa 2

A seguir constrói-se a especificação das duas primeiras rotinas **CreateS** e **IsemptyS** (fig. 2.19). Observe-se que as cláusulas **requires** e **modifies at most** não são necessárias. Simultâneamente no trait SSTACK são gerados os sorts SStack e {}, bem como, os operadores **new** e **isempty**. Visto que o sort SStack identifica o tipo que se está especificando, agora pode-se mapear para Stack escrevendo isto na cláusula **based on sort** ....

```
Type Stack                          ||
 exports CreateS, IsemptyS,         || SSTACK : trait
         PushS,PopS,Tops            || introduces
  based on sort SStack              ||  new: {} -----> SStack
        from  SSTACK.               ||  isempty : SStack ----> Bool
                                    ||  constrains
                                    ||       ........
```

```
FUNCTION CreateS( s: Stack) : Stack
     ensures  ( s_post = new() ) &  isempty(new())  [ s ]

FUNCTION IsemptyS(var s : Stack) : Boolean
     ensures if isempty(s) then true
                           else false
PROCEDURE PushS( s : Stack, e: StackItem)
     requires .........
     modifies at most ......
     ensures ..........
FUNCTION PopS(var s : Stack) : Stack
     requires .......
     modifies at most  .........
     ensures  .......
FUNCTION TopS(var s: Stack) : StackItem
     requires ........
     modifies at most ........
     ensures ........
```

ig. 2.19 Exemplo de Especif. em Larch/Pascal: Etapa 3

O passo seguinte a CreateS e IsemptyS é especificar o corpo da rotina PushS. Dois operadores são necessários: push e top.

```
Type Stack
    exports CreateS, IsemptyS,
            PushS,PopS,Tops
    based on sort SStack
          from  SSTACK.
```

```
FUNCTION CreateS( s: Stack) : Stack
     ensures  ( s_post = new() ) &  isempty(new())  [ s ]
FUNCTION IsempyS(var s : Stack) : Boolean
     ensures if isempty(s) then true
                           else false
```

```
PROCEDURE PushS( s : Stack, e: StackItem)
     modifies at most [ s, top(s) ]
     ensures  s_post = push(s,e)
FUNCTION PopS(var s : Stack) : Stack
     requires .......
     modifies at most  .........
     ensures  .......
FUNCTION TopS(var s: Stack) : StackItem
     requires ........
     modifies at most ........
     ensures ........
 SSTACK : trait
    introduces
     new: {} -----> SStack
     isempty : SStack ----> Bool
     push : SStack, StackItem --------> SStack
     top : SStack --------> StackItem
    constrains
         top(push(stk,e)) = e
          pop(push(stk,e)) = stk
```

Fig. 2.20 Exemplo de Especif. em Larch/Pascal: Etapa 4

Na fig. 2.20 observa-se que os operadores **new** e **push** tem signaturas semelhantes com as signaturas dos operadores *new* e *insert* respetivamente, do trait CONTAINER, contido no núcleo da linguagem Larch Compartilhada. Utilizando então o mecanismo de renomeação, estes operadores (new e push) devem ser substituidos pela cláusula **includes CONTAINER with** ... . Finalmente completa-se a especificação das rotinas PopS e TopS com a inclusão do operador **pop**. As telas finais terão a seguinte forma:

```
Type Stack
exports CreateS, IsemptyS,
        PushS,PopS,Tops
based on sort SStack
     from  SSTACK.
```

```
FUNCTION CreateS( s: Stack) : Stack
     ensures  ( s_post = new() ) &  isempty(new())  [ s ]
FUNCTION IsemptyS(var s : Stack) : Boolean
     ensures if isempty(s) then true
                           else false
PROCEDURE PushS( s : Stack, e: StackItem)
     modifies at most [ s, top(s) ]
     ensures  s_post = push(s,e)
FUNCTION PopS(var s : Stack) : Stack
     requires isempty(s) = False
     modifies at most [ s, top(s) ]
     ensures  s_post = pop(s)
FUNCTION TopS(var s: Stack) : StackItem
     requires isempty(s) = False
     modifies nothing
     ensures StackItem = top(s)

SSTACK : trait
     includes  ENUMERABLE with [ push for insert, top for
                         last, pop for rest, SStack for C,
                         StackItem for E ]
     constrains push, pop, top so that
          for all [ stk : SStack, e: StackItem ]
          top(push(stk,e)) = e
           pop(push(stk,e)) = stk
```

Fig. 2.21 Exemplo de Especif. em Larch/Pascal: Etapa Final

O trait ENUMERABLE faz parte do núcleo da LLC e inclue os traits CONTAINER (new, insert), LAST (last), REST (rest) e ISEMPTY (isempty).

2.3 Comparação com outras linguagens de Especificação Algébrica

Considerando que as quatro linguagens descritas na Introdução deste texto são de natureza algébrica e modular, é possível apontar algumas diferenças e similariades e também algumas vantagens e desvantagens entre elas e a linguagem Larch Compartilhada.

2.3.1 Diferenças e Similaridades

a) *Sintaxe*.

i. Módulos:

Larch (trait) e OBJ (objeto) usam um único tipo de módulo como unidade básica, enquanto ABEL (type, class, group, property), ASL (module, cluster, scheme, instantiate) e Iota (type, sype, procedure, realization) utilizam mais de um tipo de estrutura modular.

ii. Sorts:

OBJ tem sorts e subsorts e uma relação entre eles.

iii. funções ou operadores:

OBJ usa atributos para operadores (associatividade, identidade, etc.). Iota utiliza funções multi-imagem, i.e., o contradomínio de uma função é a união disjunta de dois ou mais sorts.

b) *Semântica*

OBJ e ASL estão baseadas em semântica de algebras iniciais multi-sortidas. As semânticas de ABEL e Iota estão baseadas em uma versão modificada de lógica de primeira ordem com igualdade (Teoría de lógica fraca e iota-lógica, respectivamente). Larch está baseada em uma teoría de cálculo de predicados de primeira ordem tipado com equações como fórmulas atômicas.

c) *Tratamento de Erros e situações exepcionais*

Larch Compartilhada não faz tratamentos de erros. Todas as pré-condições e erros são manipulados pelas linguagens de interface Larch. Termos como *pop(newstack)* são considerados bem-formados. No entanto, existe o mecanismo exempts para indicar que o significado de pop(newstack) em um trait intencional_ mente não foi restrito pelas equações. Iota usa contradomínios multisortidos para tratar erros, como f: S1 -> S2 ∪ {erro1,...,erroN }. OBJ usa dois tipos de operadores (e equações): para situações excepcionais e para situações de erro (ERR-OPS). ASL em [HOR 88] tem uma nova abordagem: o uso de marcadores de segurança e insegurança ($, $$, !, ?, ??)

d) *Mecanismo de sorts e operadores Ocultos*

OBJ e ASL usam sorts e operadores ocultos explicitamente (com a palavra HIDDEN) e implícitamente. Larch não providencia nenhum mecanismo para sorts ou operadores ocultos. Na realidade, estes operadores não são completamente ocultos, pois devem ser lidos para poder compreender a especificação. Como sorts e operadores em Larch, são considerados auxiliares (não são implementados !!), toda a abordagem Larch

Compartilhada pode ser considerada oculta (em [CAS 88] explica-se o não uso de operadores ocultos).

e) *Parametrização*

A ideia básica de parametrização é maximizar o re-uso de um programa desenvolvendo programas tão gerais quanto possível, i.e., construir um novo módulo a partir de outros já existentes somente instanciando um ou mais parâmetros. OBJ usa três conceitos de parametrização: teorias, visões e expressões módulo. Iota também tem seu mecanismo de parametrização (Ex. POLY(P:RING) ou RING.POLY). A habilidade para substituir qualquer identificador de sort ou operador aparecendo em um trait, através da lista **with** [... **for** ..., ...] e não atraves da teoría associada ao trait, transforma a Linguagem Larch Compartilhada em uma linguagem efetivamente parametrizada sem problemas semânticos.

f) *Completeza.*

Das cinco linguagens, somente Larch tem o mecanismo que permite verificar se uma especificação é completa (completeza suficiente). A cláusula *generated by* mostra o conjunto de operadores que são suficientes para gerar todos os valores do sort que está se especificando. Larch fornece as cláusulas *converts* e *exempts* para o especificador estabelecer quais serão as propriedades de completeza que serão verificadas.

g) *Construção de especificações.*

As especificações em Larch são construídas ao redor de incrementalidade, i.e., agregar algo a um trait nunca remove fórmulas da sua teoría associada. Desta maneira usando a idéia de extensão conservativa, fica

mantida a propriedade de monotonicidade para especificações. A escolha de uma álgebra inicial ou final, como em OBJ ou ASL, implica na perda da monotonicidade.

h) *Dependência de uma linguagem de programação.*

OBJ, ABEL e IOTA são linguagens tanto de especificação como de programação, isto é, são executáveis, portanto, tem suas próprias características de programação. Larch é somente uma linguagem de especificação, mas cada módulo tem uma parte que depende diretamente da linguagem de programação acessível ao usuário (especificador ou programador).

i) *Concorrência.*

OBJ, ASL,IOTA,ABEL e Larch foram desenhadas para programas sequenciais. Guttag e Wing em [BIR 87] e [HER 86], apresentaram uma extensão da Larch Sequencial ao tratamento de processos concorrentes. Tambem ABEL tem uma extensão à concorrência através de CABEL.

j) *Uso de especificações.*

Uma especificação não é desenhada somente para conter informação, mas para comunicar esta informção de uma maneira efetiva. Portanto, a especificação deve ser clara, com uma apresentação concisa, com uma estrutura sintática que possa ser lida e compreendida com facilidade. Nem sempre uma especificação "grande" é completa. OBJ, IOTA e ABEL por serem linguagens orientadas também à programação, não satisfazem as características acima mencionadas, não são muito "amigáveis" ao usuário comum. Por outro lado, a redundância e a estrutura simples da Larch, com poucas palavras chaves e poucas cláusulas, facilita a sua

rápida e melhor compreensão.

k) *Ferramentas*.

As cinco linguagens que estão sendo comparadas, estão munidas de um conjunto de ferramentas quase suficientes: editores de texto, editores dirigidos pela sintaxe, analisadores (sintáticos e semânticos), depuradores, provadores de teoremas, etc.

2.3.2 Vantagens e desvantagens

a) *Vantagens*:

* Existe uma clara distinção sintática e semântica entre especificação de propriedades de abstrações básicas e especificações de propriedades de componentes de programas. Exemplo:

abstração:

```
remove: RelDB, Tup --------> RelDB
.....................................
remove(newR(a1,k1),t) = newR(a1.k1)
```

Procedimento:

```
PROCEDURE DelTup(R:Relation; T:Tup)
      modifies at most [ R ]
      ensures R_post = remove(R,T)
```

remove elimina um elemento t de um conjunto r e **DelTup** elimina uma tupla de uma relação.

* O conjunto de abstrações usadas para especificar interfaces é aberto (podem ser incluidos outros traits), entretanto, cada abstração é bem definida.

* Reusabilidade: especificações de abstrações podem ser fácilmente re-usadas por componentes de programas escritos em diferentes linguagens. Está disponível um conjunto de especificações básicas, a modo de Handbook. O mecanismo de renomeação, permite usar um trait de infinitas maneiras; por exemplo, o trait *Enumerable* contendo os operadores insert, next e rest, é utilizado para especificar:

  stack:

  ```
  includes Enumerable with [ push for insert,
                             top for next, pop for rest ]
  ```

  Queue:

  ```
  includes Enumerable with [ first for next ]
  ```

  Dequeue:

  ```
  includes Stack with [ enter for push, last for
                        top, prefix for pop ]
  ```

* Cada linguagem de interface pode ser otimizada para comunicar importantes propriedades de interfaces em uma particular linguagem de programação.

* *Flexibilidade*: O fato de que um trait não necessariamente corresponde a um Tipo Abstrato de Dados e a liberdade para substituir qualquer identificador de sort ou operador de um trait, fornece considerável flexibilidade ao especificador.

* *Completeza*: Um trait bem-formado não necessariamente deve ter a propriedade de completeza. Os mecanismos **converts** e **exempts** permitem estabelecer ao especificador um subconjunto de propriedades de completeza que serão verificadas posteriormente.

* *Problemas semânticos*: Considerar um trait como um simples objeto textual (sorts e operadores não são implementados) e teorías associadas como teorías de primeira ordem, permite fugir de um conjunto de problemas semânticos associados a teorías parametrizadas, visões, schemas, morfismos, etc., proprios de outras linguagens baseadas em modelos.

* *Concorrência*: Os mecanismos definidos em [HER 86] e [BIR 87] para especificações de processos concorrentes colocam a familia de linguagens Larch dentro de um espectro de aplicação muito mais amplo em relação a outras linguagens algébricas.

b) *Desvantagens*:

* As linguagens Larch não são orientadas à programação, em consequência especificações escritas em Larch não podem ser executadas.

* Ainda não é possível ter prototipação rápida através das linguagens Larch.

* São poucas as linguagens de Interface que atualmente estão bem definidas: Larch/CLU, Larch/Pascal e Larch/Ada-88. Outras como Larch/Cedar-Mesa e Larch/Modula-2 ainda estão sendo exploradas.

3. ESPECIFICAÇÃO DO MODELO RELACIONAL PARA BANCO DE DADOS

3.1 Introdução

Nesta parte apresenta-se uma aplicação da Linguagem de especificação formal LARCH. Primeiramente, na parte introdutoria aparecem algumas definições e logo o conjunto de módulos de especificações LARCH.

A Fig. 3.1 mostra o esquema da aplicação: a parte I (parte básica reusável) é um conjunto de 22 traits incorporados do Núcleo da linguagem Larch Compartilhada. A parte II, é construida sobre as definições das seções 3.1.1-3.1.6 e é a contribuição pessoal do autor neste capítulo. Compreende seis traits principais: MULTISET, LIST, TUPLE, THETA, RELATION e DATABASE que correspondem a conjunto, lista, tupla, operação theta, relação e banco de dados, respectivamente.

A estrutura principal na qual estão baseadas 4 das últimas especificações é o conjunto:

lista é um conjunto de nomes,
tupla é um conjunto de atributos,
Relação é um conjunto de tuplas,
Banco de dados relacionais é um conjunto de relações.

Aos traits acima mencionados estão associados três especificações interface (Larch/Pascal): Tuple, Relation e RDatabase.

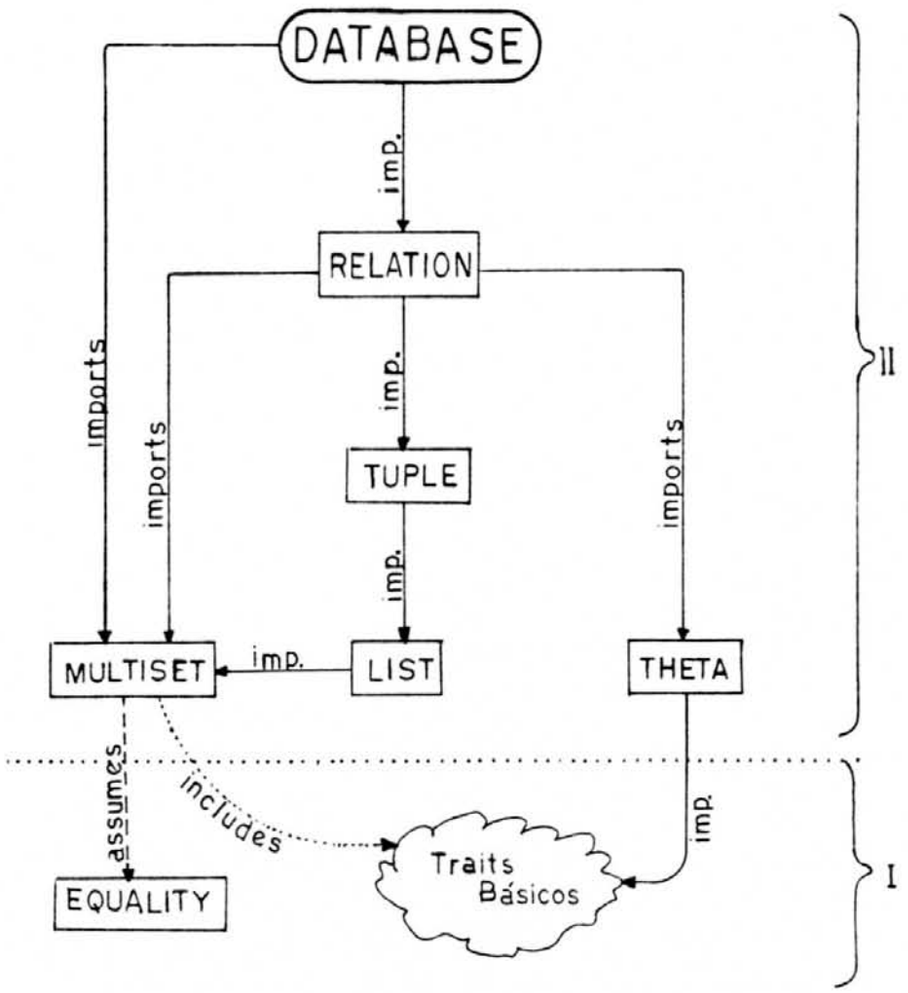

Fig. 3.1 Especificação de um BDR em LLC

Os primeiros traits estão contidos na Parte IV de [GUT 85] e constituem a base para nossas especificações (relacionadas ao modelo relacional). Cada especificação compreende três partes:

- Especificação Auxiliar (LARCH Compartilhada),
- Comentários (Especificação Informal), e
- Especificação Interface (LARCH / PASCAL).

3.1.1 Modelo Relacional

Definição ([DAT 82]). Uma descrição segundo o MODELO RELACIONAL para Dados consiste de três componentes:

- . conjuntos de objetos (domínios, atributos, chaves, tuplas, relações, etc.).
- . Um conjunto de operadores(união, diferença, produto, seleção, projeção, etc.).
- . Um conjunto de regras de integridade.

3.1.2 Sistema Relacional

Definição ([DAT 82]). Um SISTEMA RELACIONAL é um sistema de banco de dados construído de acordo com o modelo relacional, i.e., com três componentes: um banco de dados relacional, uma coleção de operações relacionais e duas regras relacionais.

3.1.3 Banco de Dados

Definição ([DAT 86]). Um BANCO DE DADOS é um conjunto de dados operacionais (valores), de alguma empresa específica, armazenados por um sistema de aplicação .

3.1.4 Banco de Dados Relacional

Definição ([MAI 85],[GEH 83]). Um BANCO DE DADOS RELACIONAL (BDR), é um conjunto finito de relações e um conjunto de operações definidas sobre estas relações.

3.1.5 Relação

Definição. Uma RELAÇAO é um conjunto finito de tuplas, e pode ser pensada como uma tabela em que as tuplas são as

filas e os componentes de cada tupla as colunas. Os componentes de uma tupla são chamados atributos, e um subconjunto destes são as chaves, usadas para identificar uma tupla. Assim, duas tuplas diferentes tem chaves diferentes.

3.1.6 Estrutura Abstrata de um Sistema Relacional

```
1. relational-database ::= domain-set  relation-set
2.              domain ::= domain-name   domain-value-set
                           ordering-indicator
3.         domain-name ::= name
4.        domain-value ::= atom
5.  ordering-indicator ::= YES | NOT
6.            relation ::= named-relation | unnamed-relation
7.      named-relation ::= real-relation | virtual-relation
8.      real-relation  ::= relation-name   attribute-set
                           primary-key  alternate-key
                           tuple-set
9.       relation-name ::= name
10.          attribute ::= attribute-name  domain-name
11.     attribute-name ::= name
12.        primary-key ::= candidate-key
13.      candidate-key ::= attribute-name-set
14.       alternate-key ::= candidate-key
15.               tuple ::= attribute-value-set
16.     attribute-value ::= attribute-name  domain-value
17.    virtual-relation ::= relation-name
                            relational-expression
18.    unnamed-relation ::= relational-expression
19.       rel-operation ::= rel-alg-operation  |
                                    rel-assignment
```

```
20.   rel-alg-operation ::= union | difference | product |
                              theta-selection | projection
21.               union ::= UNION rel-name rel-name corresp
22.             corresp ::= attribute-name-pair-set
23. attribute-name-pair ::= attribute-name  attribute-name
24.          difference ::= DIFFERENCE    relation-name
                                  relation-name   corresp
25.             product ::= PRODUCT   relation-name
                                          relation-name
26.     theta-selection ::= THETA-SELECT   relation-name
                                       theta-comparison
27.    theta-comparison ::= attribute-name theta comparand
28.               theta ::= < | ≤ | > | ≥ | = | ≠
29.           comparand ::= atom | attribute-name
30.          projection ::= PROJECT    relation-name
                                       attribute-name-set
31.      rel-assignment ::= ASSIGN    relation-name
                            relational-expression   corresp
32.relational-expression::= relation-name | rel-literal |
                                 rel-alg-operation
33.         rel-literal ::= attribute-set   tuple-set
```

Uma descrição detalhada das produções acima pode ser encontrada em [DAT 82]

3.2 Traits Básicos para especificar um BDR

a) Propriedade de Igualdade

```
1.  EQUALITY : trait
2.      introduces
3.          = : T, T --------> Bool
4.      asserts
5.          T partitioned by [ = ]
6.          for all [ t1, t2, t3 : T ]
7.              =(t1,t1)
8.              =( =(t1,t2), =(t2,t1) )
9.              ( =(t1,t2) & =(t2,t3) ) ==> =(t1,t3)
```

b) Relação Binaria

```
1. RELATIONBIN : trait
2.      introduces
3.          rb : T, T --------> Bool
```

c) Relacão Total

```
1. TOTALRELATION : TRAIT
2.      includes RELATIONBIN
3.      asserts for all [ t1,t2 : T ]
4.          or( rb(t1,t2), rb(t2,t1) )
```

d) Relação Binaria Reflexiva

```
1. REFLEXIVE : trait
2.      includes RELATIONBIN
3.      asserts    for all [ t:T ]
4.                 rb(t,t) = true
```

e) Relação Binaria Transitiva

```
1. TRANSITIVE : trait
2.      includes RELATIONBIN
3.      asserts  rb so that for all [ t1,t2,t3:T ]
4.            and(rb(t1,t2), rb(t2,t3)) ==> rb(t1,t3) = true
```

f) Relação Binaria Reflaxiva-Transitiva

```
1. REFLEXIVETRANSITIVE : trait
2.      includes REFLEXIVE, TRANSITIVE
```

g) Relação Binaria Simétrica

```
1. SYMMETRIC : trait
2.      includes RELATIONBIN
3.      asserts   for all [ t1,t2:T ]
4.                rb(t1,t2) = rb(t2,t1)
```

h) Relação Binaria de Equivalência

```
1. EQUIVALENCE : trait
2.      includes  REFLEXIVETRANSITIVE with [ eq for rb ],
                  SYMMETRIC with [ eq for rb ]
```

i) Relação de Ordem Parcial

```
1. PARTIALORDERED : trait
2.      imports REFLEXIVETRANSITIVE with [ ≤ for rb ]
```

j) Relação de Ordem Total

```
1. TOTALORDER : trait
2.     includes PARTIALORDER, TOTALRELATION with [≤ for rb ]
```

k) Relação Binaria de Ordem Parcial e de Equivalência

```
1. ORDEREQUIVALENCE : trait
2.      assumes PARTIALORDER
3.      introduces
4.           eq : T, T --------> Bool
5.      constrains eq so that for all [ x, y : T ]
6.           eq(x,y) = ≤(x,y) & ≤(y,x)
7.      implies EQUIVALENCE
8.      converts [ eq ]
```

l) Relação de Equivalência,Ordem Parcial e Igualdade

```
1. ORDEREQUALITY : trait
2.      assumes PARTIALORDER
3.      includes  EQUALITY,
4.                ORDEREQUIVALENCE with  [ = for eq ]
```

m) Relação Binaria de Ordem Parcial com Igualdade

```
1. PARTIALORDERWITHEQUALITY : trait
2.      includes PARTIALORDER, ORDEREQUALITY
```

n) Relação Binaria de Ordem Total e Igualdade

```
1. TOTALORDERWITHEQUALITY : trait
2.      includes TOTALORDER, ORDEREQUALITY
```

o) Relação Binaria de Ordem Parcial (derivadas)

```
1. DERIVEDORDERS : trait
2.      assumes PARTIALORDER
```

```
3.      introduces
4.           < : T, T --------> Bool
5.           ≥ : T, T --------> Bool
6.           > : T, T --------> Bool
7.      constrains <, ≥, > so that for all [ x, y : T ]
8.           <(x,y) = ≤(x,y) & not( ≤(y,x) )
9.           ≥(x,y) = ≤(y,x)
10.          >(x,y) = <(y,x)

11.     implies   TRANSITIVE with [ < for rb ],
                  TRANSITIVE with [ > for rb ],
                  PARTIALORDER with [ ≥ for ≤ ]
12.     converts [ <, ≥, > ]
```

p) Ordem Parcial

```
1. PARTIALLYORDERED : trait
2.      imports  PARTIALORDERWITHEQUALITY
3.      includes DERIVEDORDERS
4.      implies PARTIALORDERWITHEQUALITY with [ ≥ for  ≤ ]
```

q) Relação Binaria com Ordem

```
1. ORDERED : trait
2.      imports  TOTALORDERWITHEQUALITY
3.      includes DERIVEDORDERS
4.      implies   PARTIALLYORDERED,
                  TOTALORDERWITHEQUALITY with [ ≥ for  ≤ ]
```

r) Ordinal

```
1. ORDINAL : trait
2.      includes ORDERED with [ Ord for T ]
```

```
3.      introduces
4.           first : {} --------> Ord
5.           succ  : Ord --------> Ord
6.      asserts
7.           Ord generated by [ first, succ ]
8.           Ord partitioned by [ ≤ ]
9.           for all [ x,y : Ord ]
10.               ≤( first(), X )
11.               ≤(  succ(x), first() ) = False
12.               ≤( succ(x), succ(y) ) = ≤( x, y )
13.     converts [ =, ≤, <, ≥, > ]
```

s) Cardinal

```
1. CARDINAL : trait
2.      imports  ORDINAL with [0 for first, Card for Ord ]
3.      introduces
4.           one : {} --------> Card
5.            + : Card, Card --------> Card
6.            * : Card, Card --------> Card
7.            - : Card, Card --------> Card
8.      constrains one, +, *, - so that for all [ x,y : Card ]
9.           one() = succ(0())
10.           +(x,0()) = x
11.           +( x, succ(y) ) = succ( +(x,y) )
12.           *( x, 0() ) = 0()
13.           *( x, succ(y) ) = +( x, *(x,y) )
14.           -( 0(), x ) = 0()
15.           -( x, 0() ) = x
16.           -( succ(x), succ(y) )  = -(x,y)
17.     implies Card generated by [ one, + , - ]
                Card partitioned by [ ≥ ]
                Card partitioned by [ = ]
```

```
                Card partitioned by [ < ]
                Card partitioned by [ > ]
18.     converts [ one, -, +, *, =, ≥, ≤, >, < ]
```

t) Container

```
1. CONTAINER : trait
2.     introduces
3.         new : {} --------> C
4.       insert : C, E --------> C
5.     constrains C so that
6.         C generated by [ new, insert ]
```

u) Número de elementos

```
1. SIZE : trait
2.     assumes  CONTAINER
3.     imports  CARDINAL
4.     introduces
5.         size : C --------> Card
6.     constrains  size   so that
7.                 size(new()) = 0()
```

w) Teste de Vazio (sem elementos)

```
1. ISEMPTY : trait
2.     assumes CONTAINER
3.     introduces
4.         isempty : C --------> bool
5.     constrains isempty, new, insert so that
6.         for all [ c:C ; e:E ]
7.           isempty(new())                 % = true %
8.           not( isempty(insert(c,e)))     % = true %
9.     implies converts [ isempty ]
```

3.3 Especificação de um Conjunto Multi-uso

a) Conjunto com Repeticões (Especif. em LLC)

```
1. MULTISET : trait
2.      assumes EQUALITY with [ E for T ]
3.      includes ISEMPTY, SIZE,
                 CONTAINER with [ MSet for C ]
4.      introduces
5.               count : MSet, E --------> Card
6.          numElements : MSet  --------> Card
7.               delete : MSet, E --------> MSet
8.             contains : MSet, E --------> Bool
9.               subset : MSet, MSet --------> Bool
10.               equiv : MSet, MSet --------> Bool
11.               union : MSet, MSet --------> MSet
12.                diff : MSet, MSet --------> MSet
13.              inters : MSet, MSet --------> MSet
14.     constrains MSet so that
                    MSet partitioned by [ count ]
15.     for all [ s, s1, s2 : MSet; e1, e2 : E ]
16.          insert( insert(s, e1), e2) = if (e1=e2)
                       then insert(s, e1)
                       else insert( insert(s, e2), e1)
17.          count(new(),e1) = 0()
18.          count(insert(s,e1),e2) = count(s,e2) +
                          ( if (e1 = e2) then one()
                                         else 0() )
19.          size(insert(s,e1)) = size(s) + one()
20.          numElements(new()) = 0()
21.          numElements(insert(s,e1)) = numElements(s) +
                       ( if count(s,e1) > 0() then 0()
                                              else one() )
```

```
22.        delete(new(), e1) = new()
23.        delete(insert(s,e1), e2) = if (e1 = e2)
                                      then  s
                                      else insert(delete(s,e2), e1)

24.        contains(new(),e1) = false
25.        contains(insert(s,e1), e2) = if (e1 = e2)
                                        then true
                                        else contains(s,e2)
26.        subset(new(),s) = true
27.        subset(insert(s,e),new()) = false
28.        subset(insert(s1,e1), s2) = if contains(s2,e1)
                                       then subset(s1,s2)
                                       else false
29.        equiv(s1,s2) = subset(s1,s2) & subset(s2,s1)
30.        union(s,new()) = s
31.        union(s, insert(s1,e1)) = insert(union(s,s1),e1)
32.        diff(s,new()) = s
33.        diff(new(),s) = new()
34.        diff(insert(s1,e1), s2) = if contains(s2,e1)
                                     then diff(s1,s2)
                                     else insert( diff(s1,s2), e1)
35.        inters(s,new()) = new()
36.        inters(s1,s2) = diff(s1, diff(s1,s2) )
37.      implies  converts [ isempty,  count,  delete,
                             numElements, size, union,
                             equiv, diff, inters ]
```

b) Comentários

linha 2:

Nas linhas 16,18,23 e 25 usamos o termo "e1=e2" para expressar a igualdade de dois elementos de um conjunto s. Dependendo do caso, e1 e e2 podem ser inteiros,

reais, conjuntos simples, tuplas, relações, etc.. Para inteiros ou reais o significado de "e1=e2" é obvio e não existe ambiguidade, mas no caso, de conjuntos ou relações por exemplo, "e1=e2" é confuso. Isto se resolve descarregando a teoría (axiomas) para "=" especificada no trait EQUALITY.

linha 3: INCLUSÕES

nesta linha se inclui termos como:

isempty(s), isempty(insert(s,e)), etc.
size(new())
insert(new(),e), insert(insert(s,e1),e2), etc.

A escolha de **includes** por **imports** deve-se a mudanca em CONTAINER: *MSet for C* e a alteracão do operador size (na linha 19).

linhas 4-13: OPERADORES

count(s,e): retorna o *número de vezes* que o elemento e é inserido no conjunto s.
size(s): retorna o *tamanho* do conjunto s, i.e., o número de elementos inseridos em s, incluindo elementos repetidos.
numElements(s): devolve o *número de elementos* contidos no conjunto s (sem repetições).
delete(s,e): retira o elemento e do conjunto s, i.e., retorna outro conjunto sem o elemento e.
contains(s,e): verifica se o elemento e pertence ao conjunto s.
subset(s1,s): verifica se o conjunto s1 está contido no conjunto s.
equiv(s1,s2): verifica se os conjuntos s1 e s2 são

equivalentes (s1 e s2 tem os mesmos elementos).

union(s1,s2): retorna outro conjunto correspondendo a união dos conjuntos s1 e s2.

diff(s,s1): devolve o conjunto diferênça: s - s1.

inters(s1,s2): retorna o conjunto interseção de s1 e s2.

linha 14:

A cláusula "MSet partitioned by [count]" indica que somente o operador count é suficiente para distinguir dois termos distintos do sort MSet, i.e., se para cada termo u, count(t1,u) = count(t2,u) então t1=t2.

Por exemplo:

Se

```
count(insert(s1,e1),e) =
                count(insert(insert(new(),e2),e1),e)
```

então

```
insert(s1,e1) = insert(insert(new(),e2),e1).
```

linha 37:

Os operadores que aparecem nesta lista são definidos em termos de outros operadores:

```
isempty(  ) = false()
count(  ) = 0()
count(  ) = +(    )
delete(     ) = if   then    else
union(     ) = insert(     )
equiv(    ) = subset(  ) & subset(   )
```

Por tanto, para fins de verificação, todos os termos onde estes operadores aparecem (a esquerda do =), devem ser convertidos em seus equivalentes (os que aparecem a direita do =. Ex. union para insert).

3.4 Especificação de uma Lista e a Operação Theta

a) Lista

```
1. LIST : trait
2.      imports MULTISET with [ List for MSet, Name for E,
                                sublist for subset ]
```

b) Operação Theta

```
1. THETA : trait
2.    imports CARDINAL
3.    introduces
4.        theta : V, V, OpBin --------> Bool
5.    constrains theta  so that
6.          for all [ v1,v2:V; EQ,NE,GT,LT,GE,LE: OpBin ]
7.         theta(v1,v2,EQ) = (if =(v1,v2) then true
                                           else false )
8.         theta(v1,v2,NE) = (if =(v1,v2) then false
                                           else true  )
9.         theta(v1,v2,GT) = (if >(v1,v2) then true
                                           else false )
10.        theta(v1,v2,LT) = (if <(v1,v2) then true
                                           else false )
11.        theta(v1,v2,GE) = (if ≥(v1,v2) then true
                                           else false )
12.        theta(v1,v2,LE) = (if ≤(v1,v2) then true
                                           else false )
13.   implies converts [ theta ]
```

c) Comentários

O trait THETA é importado pelo trait RELATION para especificar as operações sel, tj e ej.

3.5. Especificação de uma Tupla

a) Especificação em LLC

```
1. TUPLE : trait
2.     imports LIST
3.     introduces
4.        newT : List                 --------> Tup
5.       store : Tup, Name, Val       --------> Tup
6.       coord : Tup                  --------> List
7.     isvalue : Tup, Name            --------> Bool
8.      select : Tup, Name            --------> Val
9.        addv : Tup, Name, Val       --------> Tup
10.     equalT : List, Tup, Tup      --------> Bool
11.   projectT : Tup, List           --------> Tup
12.       cate : Tup, Tup            --------> Tup %catenate%

13.    constrains newT, store, select, addv, equalT,
                  projectT, cate so that
14.    Tup generated by [ newT, store ]
15.    Tup partitioned by [ equalT, coord ]
16.    for all [ t, t1, t2 : Tup; n, n1, n2 : Name;
                 v, v1, v2 : Val; l, l1, l2 : List ]
17.       coord( newT(l) ) = l
18.       coord( store(t,n,v) ) = coord(t)
19.       isvalue( newT(l), n) = False
20.       isvalue( store(t, n1, v), n2) = if n1 = n2
                                  then True
                                  else isvalue(t,n2)
21.       select( store(t,n2,v), n1) = if (n1=n2)
                                       then v
                                       else select(t,n1)
```

```
22.        addv(newT(l),n,v) = store(newT(l),n,v)
23.        addv(store(t,n2,v2), n1,v1) =
                              store( addv(t,n1,v1), n2,v2)
24.        equalT( l, t,newT(l) ) = if t =  newT(l)
                                    then True
                                    else False
25.        equalT( insert(l,n),t1,t2) =
                    if select(t1,n) = select(t2,n)
                       then equalT(l,t1,t2)
                     else false
26.         projectT(t,new()) = newT( coord(t) )
27.         projectT(t,insert(l,n) =
               store(projectT(t,l),n,select(t,n) )

28.         cate(newt(l1), newT(l2)) = newT(union(l1,l2))
29.         cate( newT(l), store(t,n,v)) =
                      store( cate( newT(l), t), n, v)
30.         cate( store(t1, n1, v1), store(t2,n2,v2) ) =
                    store( cate(t1, store(t2,n2,v2)), n1, v1)
31.      implies converts [ addv, projectT, cate ]
32.      exempts
33.      for all [t:Tup; n:Name; v1,v2:Val]
                                    addv(store(t,n,v2),n,v1)
34.      for all [l:List; n:Name] select( newT(l), n)
35.      for all [t1,t2:Tup]
              if inters( coord(t1), coord(t2) ) ≠ new()
                            then cate(t1,t2)
```

b) Comentários

linha 2:

Ao importar LIST estamos considerando cada elemento Tup como um conjunto de atributos (nomes). LIST permite tratar um elemento do sort Tup simplesmente como um

conjunto qualquer, independente das propriedades de Tup.

linhas 3-12:

newT(l) : Cria uma nova tupla(vazia) com uma lista l de atributos. T = (n1,n2,...,nl)

store(t,n,v) : Armazena um valor v no atributo n da tupla t. t=(n1,n2,...,v,...,nl).

coord(t) : devolve a lista completa de atributos da tupla t.

isvalue(t,n) : Faz o teste da existência de um valor no atributo n da tupla t.

select(t,n) : Seleciona o valor do atributo n da tupla t v = select(t,n).

addv(t,n,v) : Igual que store, exceto quando o atributo n já tem um valor.

equalT(l,t1,t2) : Verifica se duas tuplas t1 e t2 são iguais (i.e., t1 e t2 tem os mesmos valores para a lista de atributos l).

projectT(t,l) : Extrai de t uma nova tupla com um conjunto de atributos l.

cate(t1,t2) : Concatena duas tuplas t1 e t2.

linha 14:

Os geradores do sort Tup são newT e store.

Qualquer termo contendo os operadores addv, projectT e cate podem ser expressos usando somente newT e store. Esta cláusula permitirá verificar se o trait TUPLE tem completeza suficiente.

linha 15:

Dois elementos t1 e t2 do sort Tup são iguais se e somente se:

equalT(l,t1,t2) = true e

coord(t1) = coord(t2)

Observar que para:

t1 = (n1,n2,"alfa",n4),

coord(t1) = { n1,n2,n3,n4}

t2 = (n1,n2,n4)

coord(t2) = { n1,n2,n4} e

L = { n1, n2, n4 }

tem-se que equalT(L,n1,n2) = true, logo t1 = t2. Incorreto ...! , pois coord(t1) ≠ coord(t2). A diferença entre dois termos t1 e t2 do sort Tup, só pode ser observada usando os operadores observadores equlaT e coord.

linha 31:

Os operadores addv, projectT e cate são definidos em termos de outros operadores (new, store), por tanto, no processo de verificação (automática), cada termo (expressão) de addv, projectT e cate, primeiramente deveriam ser convertidos em seus equivalentes.

linha 32:

Existem três situações de exceção no trait TUPLE:

- Não é possível armazenar um valor em um atributo que já possui outro valor armazenado.
- Não é possivel selecionar um valor de uma tupla vazia.
- A concatenação de duas tuplas só é possivel se não tem atributos comuns. Ex. Se

t1 = (n1,"a",n3,"b") l = { n1, n2, n3, n4 }

t2 = (n1,"c","d") l = { n1, n4, n5 }

então cate(t1,t2) é impossível.

c) Especificação Interface (Larch/Pascal)

```
TYPE   Tuple
    exports CreateTup, StoreVal, Addv,
            EqualTup, SelectVal, ProjectTup,
            Catenate
    based on sort  Tup
          from TUPLE

FUNCTION CreateTup( l:List ) : Tuple
    requires l ≠ new()
    ensures  CreateTup = newT(l)

PROCEDURE StoreVal( t:Tuple; n:Name; v:Val)
    requires contains( coord(t), n) = True
    modifies at most [ t ]
    ensures t_post = store(t,n,v)

PROCEDURE Addv(t:Tuple; n:Name; v:Val )
    requires isvalue( t_pre ,n) = False
    modifies at most  [ t ]
    ensures t_post = addv(t,n,v) && isvalue(t_post,n) = true

FUNCTION EqualTup( l:List; t1,t2:Tuple ) : Boolean
    requires  l = coord(t1) = coord(t2)
    modifies nothing
    ensures EqualTup = equalT(l,t1,t2)

FUNCTION SelectVal(t:Tuple; n:Name ) : Val
    requires isvalue(t,n) = true    &&
             t_pre = newT( coord(t) )
    modifies nothing
    ensures SelectVal = select(t,n)
```

```
FUNCTION ProjectTup( t:Tuple; l:List ) : Tuple
     modifies nothing
     ensures ProjectTup = projectT(t,l)

FUNCTION Catenate( t1,t2:Tuple ) : Tuple
     requires inters( coord(t1), coord(t2) ) = new()
     modifies nothing
     ensures catenate = cate(t1,t2)
```

3.6 Especificação de uma Relação

a) Especificação em LLC

```
1. RELATION : trait
2.  imports CONTAINER with [RelDB for C, Tup for E ],
            ISEMPTY with [RelDB for C ],
            MULTISET with [ RelDB for MSet, Tup for E ],
            TUPLE, THETA
3.     introduces
4.       newR : List, List --------> RelDB
5.     create :  List, List --------> RelDB
6.        att : RelDB --------> List
7.       keys : RelDB --------> List
8.       addt : RelDB, Tup --------> RelDB
9.       addl : RelDB, Tup --------> RelDB
10.    remove : RelDB, Tup --------> RelDB
11.       sel : RelDB, Name, Val, OpBin ----> Tup
12.   product : RelDB, RelDB -----> RelDB
13.   project : RelDB, List --------> RelDB
14.        tj : RelDB, Name, RelDB, Name, OpBin
                                    --------> RelDB
15.        ej : RelDB, Name, RelDB, Name
                                    --------> RelDB
16.       div : RelDB, RelDB --------> RelDB

17.  constrains newR, create, att, keys, addt, addl,
                remove, project, tj, ej, div so that
18.   RelDB  generated  by [ newR, addt, addl, remove,
                              project, tj, ej, div ]
19.  RelDB partitioned by [ att, keys,sel, equalT ]
20.  for all [ r1,r2,r:RelDB; t,t1,t2:Tup; op: OpBin;
               k1,k11,k12,a1,a11,a12:List; n1,n2:Name;
               v:Val ]
```

```
21.    newR(a1,k1) = new()
       create(a1,k1) = newR(a1,k1)
       att( newR(a1,k1) ) = a1
       att( insert(r,t) ) = att(r)
25.    keys( newR(a1,k1) ) = k1
       keys( insert(r,t) ) = keys(r)
       addt( newR(a1,k1), t) = insert( newR(a1,k1), t)
       addt( insert(r,t1), t2) = insert( addt(r,t2), t1)
       add1( newR(a1,k1), t) = insert( newR(a1,k1),t)
30.    add1( insert(r,t2), t1) =
                        if equalT( att(r),t1,t2)
                           then  insert(r,t2)
                           else insert( add1(r,t1), t2)
31.    remove(newR(a1,k1),t) = newR(a1,k1)
32.    remove(insert(r,t2), t1) = if equalT(keys(r),t1,t2)
                                  then r
                                  else insert( remove(r,t1), t2)
33.
       sel(newR(a1,k1),n,v,op) = if contains(a1,n)
                                 then newT(a1)
                                 else newT( new() )
34.
       sel(insert(r,t),n,v,op) =
           if theta(select(t,n),v,op) & equiv(att(r),a1)
                      then t
                      else sel(r,n,v,op)
35.
   project(newR(a11,k1), a12) =  newR(a12,new())
36.
   project( insert(r,t), a1) =
                     add1(project(r,a1), projectT(t,a1) )
37.
   product(newR(a1,k1),r) =
                         newR(union(a1,att(r)),k1)
```

```
38.
product(insert(newR(a1k11),t1),insert(newR(a12,k12),t2))=
          insert(newR(union(a11,a12),k11),cate(t1,t2)
39.
product(insert(r1,t1),insert(r2,t2) ) = union(
         insert( product(insert(r1,t1),r2), cate(t1,t2)),
                  product(r1,{t2})    )
40.  tj(newR(a1,k1), n1, r, n2, op ) = newR(a1,k1)
41.  tj(r, n1, newR(a1,k1), n2, op ) = new(a1,k1)
42.
tj(insert(r1,t1),n1,insert(r2,t2),n2,op) =
   if theta(select(t1,n1), select(t2,n2),op)
   then union(
        insert(tj(r1,n1,insert(r2,t2),n2,op), cate(t1,t2)) ,
        tj( insert(r1,t1),n1,r2,n2,op)    )
   else union(
        tj(r1,n1,insert(r2,t2),n2,op) ,
        tj(insert(r1,t1),n1,r2,n2,op)    )
43.  ej(new(), n1, r, n2, op ) = new()
44.  ej(r, n1, new(), n2, op ) = new()
45.
ej(insert(r1,t1),n1,insert(r2,t2),n2 ) =
   if select(t1,n1) = select(t2,n2)
   then union(
        insert(ej(r1,n1,insert(r2,t2),n2 ), cate(t1,t2)) ,
        ej( insert(r1,t1),n1,r2,n2 )    )
   else union(
        ej(r1,n1,insert(r2,t2),n2 ) ,
        ej(insert(r1,t1),n1,r2,n2 )    )
46.
div(newR(a1.k1),r) = newR(diff(a1,att(r)),diff(k1,att(r)) )
47.   div(r,newR(a1,k1)) = project( r, diff( att(r), a1) )
```

```
48.
div( insert(r1,t1),insert(r2,t2) ) =
 if sublist( att(r2),att(r1) )
 then ( if equalT(att(r2),t1,t2)
        then inters(insert(div(r1,insert(r2,t2)),
                           project(t1,diff(att(r1),att(r2)))),
                    div( insert(r1,t1), r2)  )
        else div( r1, insert(r2,t2) )    )
  else newR(diff(att(r1),att(r2)), diff(keys(r1),keys(r2)) )

49.  implies  converts [ create, addt,  addi,  remove,  sel,
                         project, product, tj, ej, div ]
50. exempts
   for  all [al,kl:List]  if not( sublist(kl,al) )
                            then create(al,kl)
   for all [t1,t2:Tup; r:RelDB]  if equal(keys(r),t1,t2)
                                 then addt(insert(r,t1),t2)
   for all [al1,al2,kl:List] if not( sublist(al2,al1))
                             then project(newR(al1,kl),al2)
   for      all  [ r1,r2 : RelDB ]
            if not(equiv(att(r1),att(r2))) then
                  union(r1,r2), diff(r1,r2), inters(r1,r2)
```

b) Comentários

linha 2: IMPORTAÇÕES

- Com a importação de CONTAINER (usando o mecanismo de renomeação), estamos incluindo os operadores new e insert.

- Com a importação de ISEMPTY pode-se verificar se uma relação r é vazia. Termos como :

```
isempty(r),
isempty(newR(al,kl)),
isempty(insert(r,t))
```

podem aparecer na inclusão ou importação de RELATION.

- Com MULTISET importa-se os operadores count, size, numElements, delete, contains, subset, equiv, union, diff, e inters.

count(r,t): conta o número de vezes que a tupla t foi inserida na relação r.

size(r): diz o tamanho (em tuplas) da relação r.

numElements(r): retorna o número de tuplas (não repetidas) contidas na relação r.

delete(r,t): permite eliminar a tupla t da relação r.

contains(r,t): verifica se a tupla t pertence a relação r.

subset(r1,r): verifica se todas as tuplas da relação r1 pertencem a relação r.

equiv(r1,r2): verifica se as relações r1 e r2 tem as mesmas tuplas.

union(r1,r2): É a operação **UNION** e retorna uma relação onde cada tupla t está em r1 ou em r2.

diff(r1,r2): É a operação **DIFERENÇA** e retorna outra relação onde cada tupla t pertence a r1 mas não a r2.

inters(r1,r2): É a operação **INTERSEÇÃO** e retorna uma relação onde cada tupla esta em r1 e r2.

linhas 4-16: OPERADORES

newR(a1,k1): Cria uma nova relação com uma lista de atributos a1 e uma lista de chaves k1.

create(a1,k1): igual que newR, excepto que **create** verifica se a lista de chaves k1 é um subconjunto da lista de atributos a1.

att(r): fornece a lista de atributos da relação r.

keys(r): fornece a lista de chaves da relação r.

addt(r,t): retorna a mesma relação r com uma tupla t a mais e verifica se r não tem uma tupla com os mesmos valores (nos atributos chaves) que t.

addi(r,t): é usada na operação **project** para assegurar que a relação projetada não tem tuplas duplicadas.

remove(r,t): retorna a mesma relação r com uma tupla t a menos.

sel(r,al,n,v): É a operação **SELEÇAO** e retorna uma tupla t com atributos al e com o valor v para o atributo n.

project(r,al): É a operação **PROJEÇAO** e retorna uma relação construída a partir de r com uma lista de atributos al.

product(r1,r2): É a operação **PRODUTO** e retorna uma relação construída de r1 e r2 onde cada tupla é a concatenação das tuplas t1 de r1 e t2 de r2. Se t1 e t2 tem atributos comuns, o operador product não é aplicado.

tj(r1,n1,r2,n2,op)
: devolve a relação THETAJOIN das relações r1 e r2 comparando seus atributos n1 e n2 respectivamente, usando o operador binario op.

ej(r1,n1,r2,n2)
: É um caso particular de thetajoin. Corresponde ao EQUIJOIN das relações r1 e r2 quando os valores dos seus respectivos atributos n1 e n2, são iguais.

div(r1,r2) : devolve a relação COCIENTE quando a relação r1 e dividida por a relação r2.

linha 18:

Existem dois conjuntos disjuntos de operadores que geram independentemente o sort RelDB: A = {new, insert} e B = {newR, addt, addi, remove, project, tj, ej, div}. O primeiro é declarado implicitamente atraves de *imports CONTAINER* e o segundo é declarado explicitamente nesta linha. Relacionando os conceitos de [CAS 88], o primeiro ou a união dos conjuntos A e B constituem um conjunto de construtores de RelDB e A é o seu conjunto Básico.

linha 49: CONVERSÕES

- Qualquer termo contendo operadores da lista [create, addt, addi, remove, sel, project, product, tj, ej, div] é definido usando outros operadores ausentes nesta lista, por exemplo:

```
create(a1,k1) = newR(a1,k1)
ej(r1,n1,r2,n2) = union( ..., ...)
```

Por tanto, é preciso que no momento da verificação, estes termos sejam convertidos (Ex. create para newR, ej para union, etc.).

linha 50: EXCEÇÕES

- Não pode ser criado uma relação com uma lista de chaves que não faz parte dos atributos. Na ausência desta exceção, a especificação é inconsistente. Vejamos porque:

```
sejam   r1 = create({a1,a2},{a3})
        r2 = create({a1,a2},{a1})
```

Obviamente, r1 e r2 são duas relações DIFERENTES (ambas

vazias, mas com chaves diferentes). Não entanto, usando o operador equiv temos:

equiv(r1,r2) = subset(r1,r2) & subset(r2,r1)
= subset(new(),r2) & subset(new(),r1)
= true() & true()
= true.

i.e., r1 e r2 são iguais (equivalentes) ... que é absurdo.

- Não pode ser realizada a operação PROJEÇAO de uma relação r, quando a lista de atributos a1, para ser projetada, não é subconjunto de att(r).
O termo:

project(newR({a1,a2,a3},{a2}), {a2,a5})

é inconsistente : o atributo a5 não faz parte da lista de atributos da relação a ser projetada.

- As operações union, diff e inters para duas relações r1 e r2, só estão definidas quando att(r1) é igual a att(r2), i.e., quando os atributos (não os valores) de r1 e r2 coincidem.

c) Especificação Interface (Larch/Pascal)

```
TYPE    Relation
        exports AttributesR, keysR, AddTup, DelTup,
                Selection, Projection, ThetaJoin,
                EquiJoin, EqualR, NumTup, ProductR,
                UnionR, DifferenceR, IntersectionR,
                DivisionR
        based on sort  Rel
              from RELATION  with [ Integer for Card ]
```

```
FUNCTION AttributesR( r: Relation ) : List
     modifies nothing
     ensures AttributesR = att(r)

FUNCTION KeysR( r:Relation ) : List
     modifies nothing
     ensures   KeysR = keys(r)    &&
               sublist(keys(r), att(r) ) = true

PROCEDURE AddTup( r:Relation; t:Tup )
     requires  contains(t1,r)   &&
               equal( keys(r), t1, t ) = False
     modifies at most [ r ]
     ensures r_post = addt(r,t)

PROCEDURE DelTup( r:Relation; t:Tup)
     modifies at most [ r ]
     ensures r_post = remove(r,t)

FUNCTION Selection(r:Relation; n:Name;
                   v:Valor; op:OpBin ) : Tup
     modifies nothing
     ensures Selection = sel(r,n,v,op)

FUNCTION Projection( r:Relation; l:List ) : Relation
     modifies nothing
     ensures Projection = project(r,l)

FUNCTION  ThetaJoin(r1,r2:Relation; n1,n2:Name;
                    op:OpBin ) : Relation
     modifies nothing
     ensures ThetaJoin = tJ(r1,n1,r2,n2,op)
```

```
FUNCTION EquiJoin(r1,r2:Relation; n1,n2:Name): Relation
     modifies nothing
     ensures EquiJoin = eJ(r1,n1,r2,n2)

FUNCTION EqualR(r1,r2:Relation) : Boolean
     modifies nothing
     ensures EqualR = equiv(r1,r2)

FUNCTION NumTup( r: Relation ) : Integer
     modifies nothing
     ensures NumTup = numElements(r)

FUNCTION ProductR(r1,r2 : Relation) : Relation
     modifies nothing
     ensures ProductR = product(r1,r2)

FUNCTION UnionR( r1,r2 :Relation ) : Relation
     requires equiv( att(r1), att(r2) ) = true
     modifies nothing
     ensures UnionR = union(r1,r2)

FUNCTION DifferenceR( r1,r2 :Relation ) : Relation
     requires equiv( att(r1), att(r2) ) = true
     modifies nothing
     ensures DifferenceR = diff(r1,r2)

FUNCTION IntersectionR( r1,r2 :Relation ) : Relation
     requires equiv( att(r1), att(r2) ) = true
     modifies nothing
     ensures IntersectionR = inters(r1,r2)

FUNCTION DivisionR( r1, r2 : RElation ): Relation
     modifies nothing
     ensures DivisionR = div(r1,r2)
```

3.7 Especificação de um Banco de Dados Relacional

a) Especificação em LLC

```
1.DATABASE : trait
2.  imports MULTISET with [ rdb for MSet, RelDB for E ],
3.          RELATION
```

b) Comentários

linha 2:

Observe que, teóricamente, é possível implementar as operações de união, diferença e interseção de banco de dados, pois o tratamento abstrato de um BDR como um conjunto faz isto viável, mas na prática, geralmente nunca acontece.

As seguintes operações são importadas de MULTISET:

```
          new() : cria um novo BDR.
  insert(rdb,r) : insere uma relação r no banco rdb.
  delete(rdb,r) : elimina uma relação r do banco rdb.
   isempty(rdb) : verifica se o banco rdb está vazio.
contains(rdb,r) : verifica se a relação r está presente
                  no banco rdb.
```

linha 3:

A maior parte da teoría do trait DATABASE está constituida pela teoría importada de RELATION, que fornece o conjunto de operações principais para manipular relações em banco de dados relacionais.

c) Especificação Interface (Larch/Pascal)

```
TYPE   RDatabase
       exports CreateDB, DropDB, CreateR,
               InsertR, DeleteR, AlteraR
       based on sort  rdb
            from DATABASE

PROCEDURE CreateDB(var D: RDatabase; name: string )
          modifies at most [ D ]
          ensures Dpost = new()

PROCEDURE DropDB(var D:RDatabase)
          modifies at most [ D ]
          esnsures Dpost = new()

PROCEDURE CreateR(var D:RDatabase; r:RelDB; al,kl:List)
          requires  contains(D,r) = false
          modifies at most [D,r]
          ensures rpost = newR(al,kl)

PROCEDURE InsertR(var D:RDatabase; r:RelDB )
          requires contains(D,r) = false
          modifies at most [ D ]
          ensures Dpost = insert(D,r)

PROCEDURE DeleteR(var D:RDatabase; r:RelDB)
          requires contains(D,r) = true
          modifies at most [ D ]
          ensures Dpost = delete(D,r)
```

```
PROCEDURE AlteraR(var D:RDatabase; r:RelDB;
                  t:Tup; a,v:string)
          requires contains(D,r) = true and
                   contains(r,t) = true
          modifies at most [ D, r, select(t,a) ]
          ensures select(t,a)post = store(t,a,v)
```

CONCLUSÕES

Neste trabalho, foram considerados dois tópicos principais: a apresentação das linguagens Larch e a sua aplicação a um problema real. Na primeira parte apresenta-se aspectos sintáticos-semânticos genéricos da Linguagem Larch Compartilhada, bem como das linguagens de Interface Larch. A escolha de Larch/Pascal deve-se à linguagem de programação Pascal ser mais familiar aos possíveis usuários deste texto.

Em relação a segunda parte do texto, outros trabalhos como [GEH 83] e [MAI 85] já apresentaram uma especificação parcial do modelo relacional, utilizando a abordagem orientada a modelos (álgebras). Parcial, pela presença de operadores ocultos e a ausência de alguns módulos.

A contribuição pessoal do autor neste trabalho resume-se nos seguintes aspectos:

- a exemplificação da maior parte dos conceitos e definições.
- a especificação do trait SSTACK (LLC) e do tipo Stack (Larch/Pascal) da seção 2.2.
- o estudo comparativo das linguagens Larch com outras linguagens algébricas (secão 2.3).
- a aplicação das linguagens Larch (especificações) das seções 3.3 - 3.7.

O objetivo de apresentar uma especificação "grande" e completa para mostrar a utilidade das linguagens Larch, acredito que foi alcançado. Os comentários que acompanham cada especificação, só servem para melhorar a sua compreensão e documentação, possibilitando uma fácil

leitura. A importancia do exemplo aqui mostrado é apenas relacionada com Especificação Formal.

Espera-se que trabalhos posteriores orientados a aplicações e implementação de ferramentas que acompanhem o desenvolvimento de especificaçoes em Larch, possam detetar outras potencialidades ou fraquezas das linguagens, assim como sugerir modificações.

ANEXO

Definições BASICAS

Sort

Um sort é um identificador (nome ou símbolo) de um objeto ou domínio de valores. Corresponde a TIPO nas linguagens de programação (C, Pascal, etc.).

Exemplos: São sorts: i, c, stack, S, T, etc.

- i identifica um inteiro.
- c identifica um caractere.
- Stack identifica uma pilha.
- S identifica um conjunto qualquer.

Operador

Um operador op (operação ou função), é uma correspondência que associa cada elemento de um conjunto chamado Domínio (Dom(op)) a um único elemento de outro conjunto chamado Contradomínio ou Imagem (Imag(op)).

```
op: Dom --------> Imag
```

Exemplos:

```
size: Table --------> Card
     Dom(size) = Table
    Imag(size) = Card
 add: Table, Index, Val --------> Table
      Dom(add) = Table X Index X Val
     Imag(add) = Table
```

Operador Monotônico

Uma preordem R, é uma relação binaria reflexiva e transitiva definida sobre um conjunto não vazio S. O par <S,R> é chamado **Conjunto Preordenado**.

Sejam <S1,R1> e <S2,R2> conjuntos preordenados. Um operador op : S1 --------> S2 é chamado **monotonico**, se para todo x,yϵ S1

x R1 y implica que op(x) R2 op(y).

Exemplo: f é um operador monotônico:

f: < N,$\leq$ > --------> < R,$\leq$ >

f(n) = 3.1416 * n

Signatura

Uma signatura S-sortida é um par

Sig = <S,F>

onde S é um conjunto não vazio de sorts e F um conjunto não vazio de símbolos de operadores.

Exemplo:

ST = { Table, Index, Val, Bool, Card, {} }

FT = { new, add, element, eval, isempty, size }

Sig = <ST,FT>

X-termos e termos

Seja Sig=<S,F> uma signatura S-sortida e X um conjunto S-sortido de variáveis. Um conjunto de **Sig(X)-termos** bem-formados ou simplesmente **X-termos**, é definido (indutivamente) como o menor conjunto que possui as seguintes propriedades:

a) Cada variável xϵXs, sϵS, é um X-termo de sort s.

b) cada símbolo de operador

op : {} --------> sm

é um termo de sort sm, onde opϵF.

c) Se op é símbolo de operador

op : s1 x ... x sn --------> sm

e se t1, ..., tn são X-termos de sort s1, ..., sn, respectivamente, então op(t1,...,tn) também é um X-termo de sort sm.

Um X-termo sem variáveis recebe o nome de **Sig-termo** ou simplesmente **termo**. Quando não houver confusão, X-termo e termo são equivalentes.

Exemplos: Sejam S = < stack, real, bool >

F = < emptystack,pop,push,top,isempty >

X = { Xstack, Xreal, Xbool }

- pop(push(s,r)) é um X-termo de sort Stack. .
  s é uma variável de Xstack.
  r é uma variável de Xreal.
- push(s2,3.1416) é um termo.
- isempty(s4) é um termo.

X-fórmula atômica e X-formula

Sejam Sig = <S,F> uma signatura e X um conjunto de variáveis S-sortido. Uma **Sig(X)-fórmula atômica** ou simplesmente **X-fórmula atômica**, é uma das duas expressões seguintes:

a) t1 = t2, onde t1 e t2 são X-termos do mesmo sort.

b) D(t), onde D é um predicado definido ("definedness") e t é um X-termo.

Uma **Sig(X)-fórmula**, ou simplesmente uma **fórmula** (bem-formada) é definida indutivamente como uma fórmula atômica ou sendo uma das seguintes expressões:

not F, F1 or F2, F1 and F2, F1 ==> F2,

∀sx(F),∃sx(F)

onde s é um sort, x é uma variável de Xs e F, F1 e F2 são fórmulas. Uma fórmula é chamada também **axioma**.

Exemplos: As seguintes expressões são fórmulas (axiomas):

- true
- false
- not(push(s,e) = s)
- ∀ bool x,y ((x and y) = (x or y))
- add(succ(4),8) = succ(add(4,8))

Especificação Algébrica

Seja Sig=<S,F> uma signatura. Uma **Especificação Algébrica**, denotada por SPEC, é uma uma tupla <Sig,Es>, onde Es é uma familia de equações ou axiomas

$$\{Ls=Rs \ / \ s \in S\}$$

com Ls e Rs sendo Sig(X)-termos de sort S e com variáveis em X.

Abstrações Procedurais

Uma Abstração Procedural ou simplesmente um PROCEDURE (procedimento), é um mapeamento de um conjunto de argumentos de entrada a um conjunto de argumentos (resultados) de saída, com possíveis modificações das entradas ([LIS 86]).

```
procedure : Input_arg --------> Output_result

proc nameproc( I/O arg )
     requires .....
     modifies .....
     effects  .....
```

Exemplo:

- procedure ou function em Pascal.
- function em C

Abstrações de Dados

Uma Abstração de Dado (Tipo de Dado Abstrato) é um conjunto de objetos junto com um conjunto de operações.

Abstração de Dado = < objetos, operações >

data type = typename

operations

op1 ... (abstração procedural)

..... " "

opn ... " "

BIBLIOGRAFIA

[ATR 82] ATREYA,S.K. Formal Specification of a specification library. Cambridge, MIT, Dept. of Electrical Engeneering and Computer Science, 1982. M.S. Thesis.

[BIR 87] BIRRELL,A.D.; GUTTAG,J.V.; HORNING,J.J.; LEVIN,R. Synchronization Primitives for a Multiprocessor: A formal specification. Palo Alto, DEC Systems Research Center, 1987, Report 20.

[BRO 83] BRODIE,M.L. Research Issues in Database Specification. ACM Sigmod Record, New York, 13(3): 42-45, Apr., 1983.

[BRO 88] BROSDA,V; VOSSEN,G. Update and Retrieval in a Relational Database Through a Universal Schema Interface. ACM Trans. Database Systems, New York, 13(4): 449-485, Dec., 1988.

[CAS 88] CASTRO V.,A.S. Especificação Formal: uma Abordagem Algébrica. Porto Alegre, CPGCC-UFRGS, 1988.

[CLA 85] CLAYBROOK,B.G.; CLAYBROOK,A-M.; WILLAMS,J. Defining Database Views as Data Abstractions. IEEE Trans. Soft. Eng., New York, SE-11(1): 3-14, Jan., 1985.

[DAH 86] DAHL,O-J; LANGMYHR,D.F.; OWE,O. Preliminary Report on the Specification and Programming Language ABEL. Oslo, Institute of Informatics, University of Oslo, 1986, R.R. 106.

[DAH 89] DAHL,O-J.; OWE,O. Generator Induction in order sorted algebras. Oslo, Departament of Informatics, University of Oslo, 1989, R.R. 122.

[DAT 82] DATE,C.J. A formal definition of the Relational Model. ACM SIGMOD Record, New York, 13(1): 18-29, Sept., 1982.

[DAT 86] DATE,C.J. An Introduction to Database Systems. 4.ed. Saratoga, Addison-Wesley, 1986. v. 1.

[DUC 87] DUCE,D.A.; FIELDING E.V.C. Formal Specification- A comparison of two techniques. The Computer Journal, Cambridge, 30(4):316-327, Aug., 1987.

[FOR 85] FORGAARD,R. A program for generating and analyzing term rewriting systems. Cambridge, MIT Laboratory for Computer Science, 1985, M.S.Thesis.

[GAD 88] GADIA,S.K. A Homogeneous Relational Model and Query Languages for Temporal Databases. ACM Trans. Database Systems, New York, 13(4): 418-448, Dec., 1988.

[GAR 88] GARLAND,S.; GUTTAG,J.V. An overview of LP, the Larch Prover. In: INTERNATIONAL CONFERENCE ON REWRITING TECHNIQUES AND APPLICATIONS,3,1988. Apud Ramsey,N. Developing Formally Ada Programs, INTERNATIONAL WORKSHOP ON SOFTWARE SPECIFICATION AND DESIGN, Pittsburgh, May 15-20, 1989. Proceedings. Washington, IEEE, 1989. Publicado em SIGSOFT Engineering Notes, Los Alamitos, 14(3): 257-265, May,1989. p. 264.

[GAR 89] GARLAND,S. Private comunication, 1989. Apud Ramsey,N. Developing Formally Ada Programs, INTERNATIONAL WORKSHOP ON SOFTWARE SPECIFICATION AND DESIGN, Pittsburgh, May 15-20, 1989. Proceedings. Washington, IEEE, 1989. Publicado em SIGSOFT Engineering Notes, Los Alamitos, 14(3): 257-265, May,1989. p. 264.

[GEH 83] GEHANI,N.H. Informal and formal specifications with stepwise refinement. In: SOFTWARE Engineering: Development. Berkshire, Pergamon Infotech, 1983. cap. 3 p.37-52. State of the Art. Report 11:3.

[GOG 81] GOGUEN,J.A. Completeness of many-sorted equational logic. ACM Sigplan Notices, New York, 16(7) : 24-32, Sept. 1981.

[GOG 84] GOGUEN,J.A. Parameterized Programming. IEEE Transactions on Software Engenieering, New York, SE-10 (5): 528-543, Sept., 1984.

[GOG 86] GOGUEN,J.A.; TARDO,J.J. An introduction to OBJ: A linguagem for writing and testing formal algebraic program specifications. In: SOFTWARE Specification Techniques. New York, Addison-Wesley, 1986. p. 391-419.

[GUT 78] GUTTAG,J.V. The algebraic specification of Abstract Data Types. Acta Informatica, Berlin, Vol. 10(1) : 27-52, 1978.

[GUT 85] GUTTAG,J.V.; HORNING,J.J.; WING,J.M. Larch in Five Easy Pieces. Palo Alto, DEC Systems Research Center, 1985. Report 5.

[HER 86] HERLIHY,M.P.; WING,J.M. Axioms for Concurrent Objects. Pittsburgh, DCS Carnegie-Mellon University, 1986. CMU-CS-86-154.

[HOR 88] HOREBEEK,J.L. Ivo Van, Formal specifications based on many-sorted initial algebras and their applications to software engenieering. Leuven, Departement Computerwetenschappen, Katholieke Universiteit Leuven, 1988.

[HOR 85] HORNING,J.J. Conbining algebraic and predicative especifications in Larch. In: INTERNATIONAL JOINT CONFERENCE ON THEORY AND PRACTICE OF SOFTWARE DEVELOPMENT, Berlin March 25-29, 1985. Proceedings. Berlin, Spring-Verlag, 1985. V.2: Colloquium on Software Engineering. p.12-26 Lecture Notes in Computer Science 186.

[KAP 87] KAPUR,D.; NARENDRAN,P.; ZHANG,H. On Sufficient-Completeness and Related Properties of term Rewriting Systems. Acta Informatica, Berlin, 24(4): 395-415, Aug., 1987.

[KOW 84] KOWNACKI,R.W. Semantic checking of formal specifications. Cambridge, Mass. Dept. of Electric Engeenering and Computer Science, 1984. M.S. Thesis.

[LIS 86] LISKOV,B.; GUTTAG,J.V. Abstraction and Specification in Program Development. Cambridge, The MIT Press, 1986.

[MAI 85] MAIBAUM,T.S.E. Database Instances, Abstract Data Types and Database Specification. The Computer Journal, Cambridge, 28(2): 154-161, May, 1985.

[MEN 88] MENDES, S.B.T.; De AGUIAR,T.C. Métodos para especificação de sistemas. Curitiba, III EBAI 1988.

[NAK 83] NAKAJIMA,R.; YUASA, T. The IOTA Programming System. New York, Spring-Verlag, 1983. Lecture Notes in Computer Science, 160.

[PIR 82] PIROTTE,A A Precise Definition of Basic Relational Notions and of the Relational Algebra. ACM SIGMOD Record, New York, 13(1): 30-45, Sept., 1982.

[RAM 89] RAMSEY,N. Developing Formally Ada Programs. In: INTERNATIONAL WORKSHOP ON SOFTWARE SPECIFICATION AND DESIGN, Pittsburgh, May 15-20, 1989. Proceedings. Washington, IEEE, 1989. Publicado em SIGSOFT Engineering Notes, Los Alamitos, 14(3): 257-265, May,1989.

[ROT 88] ROTH,M.A.; KORTH,H.F.; SILVERSCHATZ,A. Extended Algebra and Calculus for Nested Relational Database. ACM Trans. Database Systems, New York, 13(4): 389-417, Dec., 1988.

[TOM 80] TOMPA,F.W. A Practical Example of the Specification of Abstract Data Types. Acta Informatica, Berlin, 13(3): 205-224, Mar., 1980.

[TUR 87] TURSKI,W.; MAIBAUM,T.S.E. The Specification of Computer Programs. Wokingham, Addison-Wesley, 1987.

[WIN 83] WING,J.M. A two-tiered approach to specifying programs. Cambridge, MIT Laboratory for Computer Science, 1983. MIT-LCS-TR-299.

[WIN 87] WING,J.M. Writing Larch Interface Language Specifications. ACM Transactions on Programming Languages and Systems, New York, 9(1):1-24, Jan., 1987.

[WIN 87a] WING,J.M. A Larch Specification of the Library Problem. In: INTERNATIONAL WORKSHOP SOFTWARE SPECIFICATION AND DESIGN, 4.,1987. Proceedings., Los Alamitos. CS Press,1987. p. 34-41.

[WIR 83] WIRSING,M.; PEPPER,P.;PARTSCH,H.; DOSCH,W.;BROY,M. On Hierarchies of Abstract Data Types. Acta Informatica, Berlin, 20(1) : 1-33, Oct. 1983.

[WON 86] WONG,E.; SAMSON,W.B. The Specification of a Relational Database (PRECI) as an abstract data type and its realisation in HOPE. The Computer Journal, Cambridge, 29(3): 261-268, 1986.

[ZAC 83] ZACHARY,J.L. A Syntax-Directed Tool for constructing specifications. Cambridge, MIT Laboratory for Computer Science, 1983. MIT-LCS-TR-299.

- FIM -

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

LARCH: uma alternativa para
especificação formal

Dissertação apresentada aos Srs.

Prof. Dr. José Mauro V. de Castilho

Prof. Antônio Carlos da Rocha Costa

Prof. Dr. Daltro José Nunes

Visto e permitida a impressão.
Porto Alegre, ..30/.05../.90..

Prof. Dr. Daltro José Nunes
Orientador

Prof. Ricardo A. da L. Reis,
Coordenador do Curso de Pós-
-Graduação em Ciência da Com
putação.